Le traitement d'exceptions. Aspects théoriques et pratiques

Valérie Issarny

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1118

Programme 3
Réseaux et Systèmes Répartis

LE TRAITEMENT D'EXCEPTIONS ASPECTS THEORIQUES ET PRATIQUES

Valérie ISSARNY

Novembre 1989

RR-1118

IRISA INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473
Télécopie : 99 38 38 32

Le traitement d'exceptions Aspects théoriques et pratiques *

Exception Handling Theorical and Practical Aspects

Valérie ISSARNY
IRISA-INRIA
Campus de Beaulieu, 35042 Rennes Cedex

Publication Interne n° 497 - Octobre 1989 - 80 Pages

Résumé : Le propos de cette étude bibliographique est d'analyser les mécanismes de traitement d'exceptions dans les langages de programmation impératifs. Dans un premier temps, nous constatons qu'un mécanisme de traitement d'exceptions est nécessaire dans un langage impératif si l'on veut pouvoir écrire des applications réalistes structurées. Ceci étant, il ressort que la définition sémantique d'un mécanisme est primordiale. Nous proposons tout d'abord la définition dénotationnelle avec continuation de mécanismes simples. Ensuite, nous nous attardons sur la sémantique axiomatique de quelques mécanismes mais en nous préoccupant plus particulièrement de l'aspect preuve de programmes. Après avoir étudié les mécanismes de traitement d'exceptions dans les langages de programmation, nous nous intéressons aux traitements des exceptions dans les données. Les types abstraits algébriques permettent de caractériser de tels types de données. Enfin, nous abordons le traitement d'exceptions dans le cadre de la programmation parallèle.

Mots clés : Traitement d'exceptions, Mécanisme de traitement d'exceptions, Sémantique dénotationnelle, Continuation, Sémantique axiomatique, Preuve de programmes, Types abstraits algébriques, Langage de spécification algébrique.

Abstract : This paper provides a study of exception handling mechanisms in imperative programming languages. An exception handling mechanism is needed in an imperative programming language if one wants to write structured programs. The semantics concepts of exception handling mechanisms are investigated because the definition of a formal semantics is a crucial step in the design of an exception handling mechanism. First, a denotational definition with continuations of simple exception handling mechanisms is suggested. Then, the axiomatic semantics of some exception handling mechanisms are provided; proof of correctness of some programs using these mechanisms are given. After having studied exception handling in imperative programming languages, exception handling in algebraic abstract data types is investigated. This paper concludes with exception handling in concurrent programming world.

Key words : Exception handling, Exception handling mechanism, Denotational semantics, Continuation, Axiomatic semantics, Program correctness, Algebraic abstract data types, algebraic specification.

*Ce travail a été réalisé dans le cadre du projet INRIA-BULL GOTHIC

Le traitement d'exceptions
Aspects théoriques et pratiques

Valérie ISSARNY

Publication Interne n° 497
Octobre 1989

Table des matières

1 Introduction

Une *exception* est généralement définie comme étant un "évènement rare". Qualifier un évènement d'exceptionnel est par conséquent une notion toute relative liée au contexte où l'on se situe. L'exemple classique pour illustrer ce propos est la recherche d'un élément dans un tableau. La présence de l'élément peut être considérée comme exceptionnelle si la recherche est effectuée dans le cadre du traitement d'une déclaration (double déclaration). Elle est par contre "normale" dans le cadre du traitement de l'occurrence d'utilisation. Le terme exception est parfois rattaché au terme erreur. Une *erreur* désignant la partie d'un état qui est incorrecte [Mell77], la définition est plus précise. Mais, elle n'est pas satisfaisante car si une erreur est une exception, l'inverse n'est pas vrai. En effet, nous pensons que le terme exception doit être plus général. Par exemple, lors d'une opération d'entrée sortie, il est avantageux de pouvoir considérer la fin de fichier comme une exception. Il s'avère que le traitement d'exceptions est rejeté par les puristes. Il est souvent avancé que les seules méthodes de vérification statiques permettent de garantir que des situations exceptionnelles ne surviendront jamais puisqu'elles fournissent l'environnement dans lequel le programme satisfait les hypothèses. En réalité, les hypothèses fortes ne sont pas toujours verifiées voire parfois non vérifiables (intervention humaine) [Cris82b]. Ceci étant, nous pouvons admettre l'utilité du traitement d'exceptions mais ce n'est pour autant prouver qu'une structure de contrôle spécifique est nécessaire. La structure de contrôle, appelée *mécanisme de traitement d'exceptions* et introduite dans certains langages de programmation, permet d'exprimer que la continuation "normale" d'une opération doit être remplacée par une continuation exceptionnelle quand une exception est détectée. La définition d'un mécanisme de traitement d'exceptions est souvent considérée comme inutile et il est prétexté que la commande conditionnelle permet tout aussi bien de traiter les exceptions. Nous étudions cet aspect par la suite aussi nous ne développons pas plus ici notre argumentation. Mais, nous pouvons faire remarquer qu'un mécanisme de traitement d'exceptions permet de structurer la spécification en différenciant le service standard des n services exceptionnels ($n \geq 0$). Par ailleurs, du point de vue de l'efficacité, un mécanisme de traitement d'exceptions est préférable à l'évaluation de conditions booléennes. Enfin, il faut noter que comme pour toute structure de contrôle, un mécanisme de traitement d'exceptions se définit formellement. Par conséquent, les programmes avec traitement d'exceptions sont aussi accompagnés de règles de preuve.

L'objet de ce rapport est de présenter les mécanismes de traitement d'exceptions (MTE) et plus précisément leurs aspects sémantiques. Dans le chapitre 2, nous énumérons les caractéristiques possibles d'un mécanisme de traitement d'exceptions, puis nous donnons quelques exemples. Ceci étant, nous sommes en mesure de montrer l'utilité d'un mécanisme de traitement d'exceptions dans un langage impératif. Il est important, lorsque l'on veut proposer un mécanisme de traitement d'exceptions, d'être en mesure d'en donner une sémantique formelle. Dans le

chapitre 3, nous donnons une définition dénotationnelle de mécanismes simples de traitement d'exceptions. Dans le chapitre 4, nous présentons la sémantique axiomatique de quelques mécanismes et nous nous attardons sur la vérification des programmes contenant des exceptions. Nous consacrons le chapitre 5 aux traitements des erreurs dans les types abstraits algébriques. Ce chapitre nous permet, après voir présenté le traitement des exceptions dans les programmes, de montrer comment traiter les exceptions au niveau des données. Ensuite, nous consacrons le chapitre 6 au traitement d'exceptions dans le cadre de la programmation parallèle. Nous concluons cette étude bibliographique dans le chapitre 7.

2 Traitement d'exceptions : Intérêt de définir un mécanisme spécifique

Afin de traiter les exceptions, des mécanismes particuliers ont été introduits dans les langages de programmation. Le but de ce chapitre est de montrer leur utilité. Pour cela, nous donnons la terminologie propre à un MTE puis nous donnons des exemples de mécanismes existants. Ensuite, nous voyons que les structures de contrôles existantes ne sont pas aussi satisfaisantes qu'un MTE pour traiter les exceptions. Nous concluons par une évaluation des MTE.

2.1 Terminologie

Un MTE fait intervenir plusieurs éléments notamment la condition, le signal et le traitant de l'exception. Dans un premier temps, nous donnons une définition informelle de ces composants.

Une opération qui peut terminer dans un état exceptionnel est appelé *signalant*, lorsqu'elle détecte une exception elle doit en avertir le contexte englobant. Cette dernière action est désignée comme étant le *signal de l'exception.* Une fois informé de l'occurrence de l'exception, l'appelant de l'opération doit exécuter la séquence d'instructions associée à l'exception, appelée *traitant de l'exception.* Il s'agit de substituer la continuation exceptionnelle à la continuation standard. Nous illustrons notre propos par un exemple. Nous décrivons la syntaxe employée. Une exception est signalée à l'aide de l'instruction SIGNAL. Les traitants sont déclarés entre crochets. Le mot clé SIGNALS permet de déclarer les exceptions signalées par la procédure.

Exemple 2.1 : Considérons la fonction *factorielle*. Cette fonction peut terminer dans un état exceptionnel si le calcul produit un entier non représentable en machine ou si la valeur passée en paramètre est inférieure à zéro.

```
PROCEDURE factorielle(v:INTEGER):INTEGER SIGNALS ov, neg =
VAR i : INTEGER;
```

```
BEGIN
   IF v<0 THEN SIGNAL neg ENDIF;
   IF v=o
      THEN RETURN 1
      ELSE RETURN v*factorielle(v-1)[overflow:SIGNAL ov; ov : SIGNAL ov]
   ENDIF
END
```

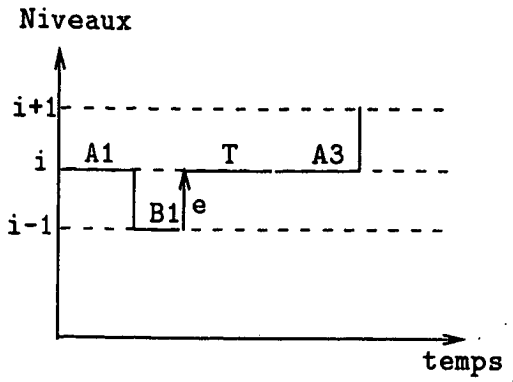
L'exception *overflow* est signalée si la multiplication provoque un dépassement de capacité; il s'agit d'une exception prédéfinie du langage. L'exception *ov* est l'exception propagée par la procédure. Lorsqu'elle est signalée, la continuation standard est abandonnée et par conséquent, la multiplication n'est pas effectuée. Enfin, l'exception *neg* est signalée lorsque la valeur passée en paramètre est négative. ◇

Lors du signal chez l'appelant, ou dans le bloc englobant, le traitant devant être exécuté est recherché dynamiquement. S'il n'existe pas de traitant dans le contexte courant, le traitant peut être recherché dans le contexte englobant. Nous parlons alors de la *propagation implicite* des exceptions. A l'inverse, la *propagation explicite*, ne permet pas d'effectuer une recherche au delà du contexte courant. (dans l'exemple 2.1, les exceptions *overflow* et *ov* sont propagées explicitement). Si le traitant de l'exception signalée ou le *traitant par défaut* ne sont pas déclarés dans le contexte courant, une exception prédéfinie *failure* est signalée. Le traitant par défaut permet de traiter l'exception *failure* ainsi que les exceptions pour lesquelles il n'existe pas de traitants explicitement déclarés. Après l'exécution du traitant, il existe deux cas de figure. Soit le MTE met en œuvre le *modèle terminaison*, soit il met en œuvre le *modèle reprise*. Dans le cas du modèle terminaison, le contrôle est définitivement rendu à l'appelant ce qui entraine un abandon définitif du signalant dès le signal de l'exception. La séquence d'instructions exécutée après le traitant est celle qui suit la liste de déclaration des traitants. Avant d'illustrer le modèle terminaison, nous étendons la syntaxe du MTE introduit en début de paragaraphe. Lorsque l'on veut associer des traitants à une séquence d'instructions, cette dernière est entre paranthèses.

Exemple 2.2 : Nous considérons les procédures A et B suivantes :

```
PROCEDURE A = BEGIN (A1; B(); A2) [e:T]; A3 END
PROCEDURE B SIGNALS e = BEGIN B1; SIGNAL e; B2 END
```

Dans la procédure A, le traitant de l'exception e est associé à la séquence d'instructions $A1; B(); A2$. L'appel de la procédure A peut être représenté par :



◇

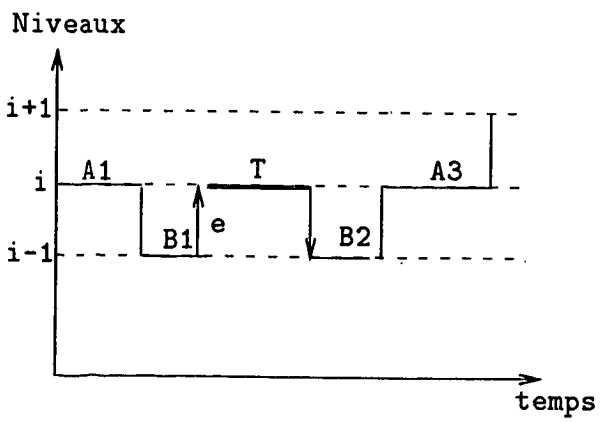
Dans le cas du modèle reprise, il existe trois options possibles. Le schéma *correction* (ou reprise) permet de reprendre l'exécution du signalant exactement après le point de signal. Le schéma *abandon* (ou terminaison) consiste à rendre le contrôle à l'appelant, l'instruction qui suit le signalant sera exécutée après le traitant. Enfin le schéma *"réessai"* offre la possibilité de réexécuter l'opération signalante. Pour plus de clarté dans notre exposé et afin d'éviter tout ambiguité, nous utilisons la notation RTR (contraction de Reprise-Terminaison-Réessai) pour désigner le modèle reprise. Le terme reprise est employé pour désigner le schéma correction du modèle RTR. Nous exemplifions le modèle RTR.

Exemple 2.3 : Nous considérons les procédures A et B suivantes :

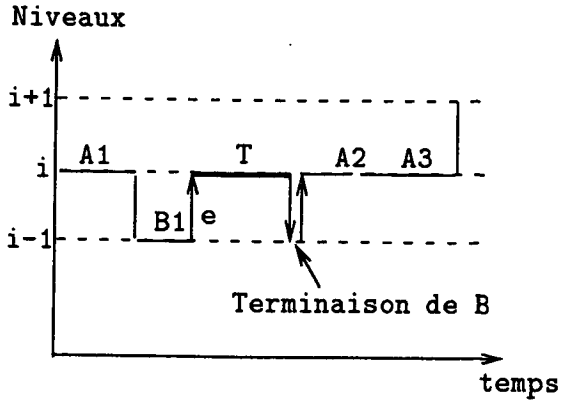
```
PROCEDURE A = BEGIN (A1; B(); A2) [e:T]; A3 END
PROCEDURE B SIGNALS e = BEGIN B1; IF cond THEN SIGNAL e ENDIF ; B2 END
```

L'appel de la procédure A peut être représenté par différents diagrammes selon l'option choisie. L'expression booléenne *cond* est supposée être évaluée à vrai.

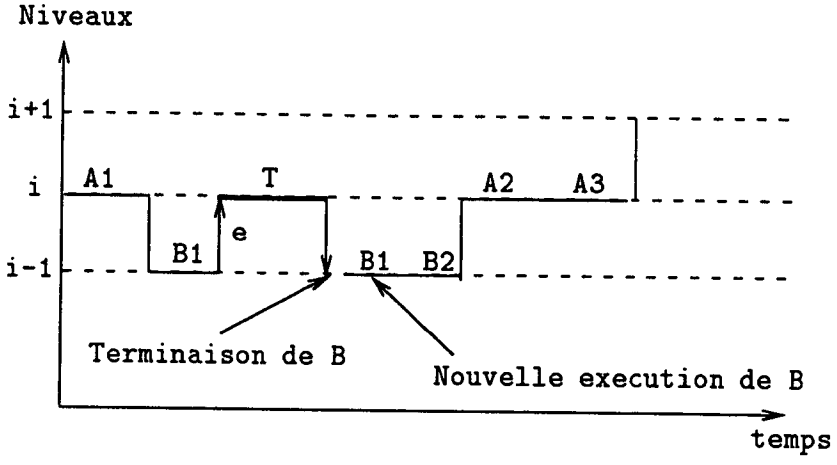
a) Reprise :



b) Terminaison :



c) Réessai où *cond* est évaluée à faux lors de la deuxième exécution de B :



◇

Nous présentons des MTE existants afin d'illustrer les différentes caractéristiques que nous venons d'exposer.

2.2 Présentation de MTE existants.

Nous prenons un exemple commun pour illustrer tous les MTE.

Exemple 2.4 : Il s'agit d'une procédure qui convertit un tableau d'entiers en une chaîne de caractères [Yemi85]. La procédure *Repr* convertit un entier en un caractère et signale l'exception *car_inexistant* s'il n'existe pas de caractère correspondant.

```
PROCEDURE Repr(INT elem_ent):CHAR SIGNALS car_inexistant;
```

La procédure *Convert* convertit le tableau d'entiers en une chaîne de caractères. Elle signale l'exception *code_incorrect* lorsqu'une valeur n'est pas convertible.

```
PROCEDURE Convert([] INT code) : STRING SIGNALS code_incorrect;
```

◇

Cette étude ne se veut pas une énumération exhaustive de tous les MTE existants dans les langages de programmation. Nous introduisons trois MTE qui nous permettent d'illustrer nos propos du paragraphe 2.1. On trouvera une étude détaillée de plusieurs MTE dans [Bana81].

2.2.1 Le MTE du langage CLU

Le MTE du langage CLU [Lisk79] met en œuvre un modèle terminaison avec propagation explicite des exceptions. Si un traitant n'est pas trouvé pour l'exception, une exception prédéfinie *failure* est signalée. Il est possible de déclarer un traitant par défaut en utilisant le mot clé OTHERS.

Nous expliquons brièvement la syntaxe du MTE. La déclaration d'une exception figure après l'instruction SIGNALS. L'instruction SIGNAL permet de signaler une exception de l'appelé vers vers l'appelant. L'instruction EXIT permet de signaler une exception localement, l'exception doit donc être traitée dans la procédure où elle est signalée. L'exécution de l'instruction SIGNAL peut être vue comme la terminaison du signalant suivie de l'exécution de l'instruction EXIT vers le traitant par l'appelant. Enfin, la déclaration des traitants se fait dans un bloc EXCEPT et la clause WHEN permet de lier un traitant à une exception. Nous présentons la programmation de l'exemple 2.4 à l'aide de ce MTE.

Exemple 2.5 : Nous donnons la procédure *convert* :

```
Convert = PROCEDURE(code : []INT) RETURNS(STRING)
SIGNALS code_incorrect(INT)
    INT i;
    STRING s := ""
    FOR i FROM LWB TO UPB code DO
        s:= s + Repr(code[i])
    END
    EXCEPT
        WHEN car_inexistant : SIGNAL code_incorrect(i)
    END
END Convert
```

L'appelant imprime la chaîne de caractères résultat :

```
DO
    ...
    PRINT(Convert(nums))
EXCEPT
    WHEN code_incorrect(i) : <corps du traitant>
END
    %Le traitement reprend ici; apres execution du traitant %
    ...
END
```

Nous présentons les différents traitants envisageables. Tout d'abord, on peut vouloir rendre une chaîne de caractères par défaut :

```
<Corps du traitant> = PRINT("")
```

On peut vouloir faire une deuxième tentative en modifiant l'élément qui est la cause de l'erreur :

```
<Corps du traitant> =
BEGIN
    nums[i] := 0; PRINT(Convert(nums));
EXCEPT
    WHEN code_incorrect(i) : PRINT("")
END
END
```

Enfin, si l'on veut que l'exception failure soit signalée dans le contexte englobant de l'appelant, l'appel peut être le suivant :

```
...;
PRINT(Convert(nums))
EXCEPT OTHERS : SIGNAL failure END
...
```

La programmation du traitant par défaut n'est pas justifiée dans ce cas. Dans la mesure où il n'y a qu'une exception signalée, il est aussi simple d'écrire "WHEN code_incorrect(i)" au lieu de "OTHERS". Par ailleurs, en omettant le traitant, on est assuré, de par la définition du MTE, du signal de l'exception *failure* dans le contexte englobant. ◇

2.2.2 Le MTE du langage MODULA-2

Le MTE du langage MODULA-2 [Rovn85], est très proche de celui du langage CLU, mais la propagation des exceptions peut être implicite. Pour ce qui est de la syntaxe, l'instruction SIGNALS de CLU est remplacée par RAISES. Les instructions SIGNAL et EXIT ont une même instruction équivalente, l'instruction RAISE. La clause TRY... EXCEPT... END permet de déclarer les traitants. La ponctuation ":" permet d'associer un traitant à une exception. Nous introduisons la nouvelle version de l'exemple 2.4.

Exemple 2.6 : Il n'est plus nécessaire de déclarer un traitant pour l'exception *car_inexistant* dans la procédure *Convert*. Les en-têtes des procédures *Repr* et *Convert* deviennent :

```
PROCEDURE Repr(elem_ent:INTEGER):CHAR;
PROCEDURE Convert(code : []INTEGER): STRING;
```

L'appelant de *Convert* s'écrit alors :

```
...
TRY
    PRINT(Convert(nums))
EXCEPT
    car_inexistant : PRINT("");
END
...
```

Il faut noter que dans ce cas, il n'existe qu'une seule solution possible pour le traitant. L'exception *car_inexistant* étant propagée, il n'est plus possible de connaître l'élément erroné dans le tableau (l'indice de l'élément erroné est connu dans la procédure *Convert* et non dans *Repr)*. ◇

2.2.3 Le mécanisme remplacement

Pour terminer, nous étudions un MTE simple mettant en œuvre le modèle RTR. Il s'agit du mécanisme remplacement [Yemi85]. Ce MTE permet toutes les réponses possibles de traitant (e.g. reprise, terminaison, réessai et propagation d'exceptions). La propagation des exceptions est explicite. Toute commande du langage peut signaler une exception du moment qu'elle la déclare. Le traitement des exceptions consiste à calculer les effets du remplacement et à retourner une valeur de remplacement. Cette valeur est destinée au signalant de l'exception s'il s'agit d'une reprise, et à l'appelant s'il s'agit d'un remplacement. La déclaration d'une exception comprend l'identificateur de l'exception, le type du paramètre pour le traitant, le type de la valeur de reprise et le type de la valeur de remplacement. Le mécanisme remplacement est destiné à des langages "orientés expression" et peut par exemple être ajouté au langage ALGOL 68. Nous donnons

la syntaxe de ce MTE. Les traitants sont déclarés entre les mots clés ON et NO. L'instruction SIGNALS permet de définir les exceptions signalées. Le signal d'une exception a la même forme syntaxique qu'un appel de procédure. Enfin, l'absence ou la présence de l'instruction REPLACE dans le corps du traitant permet de spécifier s'il s'agit d'un traitement d'exceptions avec reprise ou terminaison. La sémantique de cette instruction est simple. Lors de l'exécution de REPLACE, la valeur calculée par le traitant est retournée à l'instruction qui suit le signalant (adresse de retour du signalant). Dans le cas de la reprise (le corps du traitant se termine par END ou NO), le contrôle est rendu exactement après le point de signal. Les règles de reprise et de remplacement s'appliquent uniformément à tous les signalants, qu'il s'agisse d'une procédure ou d'une commande. Nous présentons la programmation de l'exemple 2.4 avec les différentes possibilités offertes par le MTE.

Exemple 2.7 : Nous redonnons le corps de la procédure *Convert* car les déclarations d'exceptions sont différentes. Une déclaration d'exception comprend le type du paramètre pour le traitant, le type de la valeur de remplacement et le type de la valeur de reprise. Le type de la valeur de remplacement doit être le même que celui de la valeur retournée par le signalant. Le type de la valeur de reprise doit être le même que celui de la valeur qui n'a pas été calculée lors du signal. Dans notre exemple, la valeur de remplacement pour l'exception *code_incorrect* doit être de type STRING et la valeur de reprise doit être de type CHAR.

```
PROC Convert = (REF [] INT code)STRING
SIGNALS (EXC(INT)(CHAR, STRING) code_incorrect) :
BEGIN
   STRING s := "";
   FOR i FROM LWB code TO UPB code
       DO s := s + Repr(code[i]) OD
   ON
       car_inexistant = (CHAR, CHAR) code_incorrect(i) REPLACE
   NO;
   s
END
```

L'appelant de *Convert* est de la forme suivante :

```
DO
    ...
    PRINT(Convert(nums))
    ON
    code_incorrect = (INT i)(CHAR, STRING) : <corps du traitant>
    NO
    ...
OD
```

Nous étudions les différentes réponses de traitant possibles. Si l'on désire la reprise du signalant , il suffit de donner un caractère de substitution dans le corps du traitant :

```
<corps du traitant> = '?'
```

Pour une terminaison du signalant, il faut rendre une chaîne de "remplacement", puis, déclarer qu'il s'agit d'un remplacement :

```
<corps du traitant> = "" REPLACE
```

Si l'on veut réessayer le signalant, le corps du traitant doit modifier l'élément qui est la cause de l'erreur dans le tableau d'entiers, puis "remplacer" le résultat de la fonction par un autre appel à cette fonction.

```
<corps du traitant> = BEGIN nums[i] := 0; Convert(nums); REPLACE END
```

Enfin, pour une terminaison de la construction englobante et une propagation de l'exception, il suffit de signaler une exception chez le traitant. Cette exception doit être déclarée par la construction englobante. Nous donnons la syntaxe de cette construction englobante.

```
BEGIN
   DO SIGNALS ((CHAR, VOID) fin)
      BEGIN
         ...
         PRINT(Convert(nums));
         ...
      END
      ON
         code_incorrect = (INT i)(CHAR, STRING) : fin
      NO
   OD
END
ON
   fin = (CHAR, VOID) : SKIP REPLACE
   % Le signalant est "remplace" par la valeur indefinie SKIP %
NO
```

◇

Nous venons de montrer ce qu'est un MTE mais nous n'avons pas démontré son utilité. C'est ce que nous faisons dans le paragraphe suivant.

2.3 Intérêt de définir un MTE

Nous montrons que les structures de contrôles existantes ne sont pas aussi satisfaisantes qu'un MTE pour traiter les exceptions.

2.3.1 Traitement d'exceptions à l'aide de conditionnelles

Pour programmer l'exemple 2.4 sans MTE, il suffit d'introduire un certains nombres de variables et de tests. La solution consiste à introduire une variable en paramètre du signalant pour toute exception déclarée, nous l'appelons *variable d'exception*. Il s'agit d'une variable booléenne qui est égale à vrai lorsque l'exception qui lui correspond survient. Après l'exécution d'un signalant, il faut tester toutes les variables d'exceptions. La détection des exceptions au moyen de tests interdit le modèle reprise puisque le signalant est terminé. Nous donnons une nouvelle version de l'exemple 2.4.

Exemple 2.8 : Il faut introduire un nouveau paramètre pour les procédures *Convert* et *Repr* :

```
PROC Repr = (INT elem_ent; VAR BOOLEAN car_inexistant) CHAR

PROC Convert = ([] INT code; VAR code_incorrect:BOOLEAN) STRING :
   STRING s := "";
   BOOLEAN car_inexistant := FALSE;
   INT i := LWB code - 1;
   WHILE i<UPB code AND NOT(car_inexistant) DO
      i:=i+1;
      CHAR c := Repr(code[i], car_inexistant);
      IF NOT(car_inexistant) THEN s:=s+c END;
   END;
   code_incorrect := car_inexistant;
   s
END;
```

Nous n'exposons pas le traitement de l'exception *code_incorrect* chez l'appelant. Il consiste uniquement à tester la variable d'exception puis à exécuter une séquence d'instructions correspondant au traitant déjà présenté. ◇

Ce premier exemple montre clairement que la présence d'un MTE permet de mieux structurer les applications avec occurrences d'exceptions. De plus, si l'on veut pouvoir exécuter à nouveau la procédure, il faut introduire un troisième paramètre dans la procédure *Convert* qui rend l'indice de l'élément erroné dans le tableau. De manière générale, on obtient très vite un grand nombre de variables et de tests dans les applications, et ces dernières deviennent rapidement illisibles.

2.3.2 Traitement d'exceptions à l'aide de procédures

Nous avons vu que dans le mécanisme remplacement, la syntaxe du signal d'une exception est identique à celle de l'appel de procédure. Il est donc intéressant de voir si le seul concept de procédure ne suffirait pas à traiter les exceptions d'une manière structurée. La solution consiste à appeler une procédure qui correspond au traitant et qui est passé en paramètre par l'appelant. Nous appelons

cette procédure, la *procédure traitante*. Dans un premier temps, nous présentons une mise en œuvre de l'option reprise du modèle RTR. Cette première approche est la plus simple car l'exécution reprend exactement à la suite de l'appel de la procédure traitante. Nous supposons un langage où les procédures sont passées en paramètre avec lien statique. Nous reprenons la solution avec reprise de l'exemple 2.7.

Exemple 2.9 : Nous rappelons qu'il s'agit d'ajouter un point d'interrogation dans la chaîne de caractères résultat lorsque l'élément du tableau n'a pas de caractère qui lui correspond. La procédure *Repr* admet un paramètre supplémentaire qui est la procédure traitante :

```
PROC Repr = (INT elem_ent; PROC trait_car_inexistant) CHAR
```

La procédure *Convert* a aussi la procédure traitante comme seul paramètre supplémentaire. La procédure traitante que l'on fournit à la procédure *Repr* est celle donnée à la procédure *Convert* :

```
PROC Convert = (REF [] INT code; PROC trait_code_incorrect) STRING :
   STRING s := "";
   FOR i FROM LWB TO UPB code DO
      s:=s+Repr(code[i], trait_code_incorrect);
   END;
   s;
END;
```

La procédure *trait_code_incorrect* est très simple puisqu'elle consiste à retourner le caractère '?' :

```
PROC trait_code_incorrect = () CHAR :
   CHAR c := '?'; c
END;
```

◇

Nous avons présenté une solution à l'option reprise du modèle RTR. Nous considérons maintenant le modèle terminaison car son schéma de contrôle est plus général que celui de l'option terminaison du modèle RTR; la séquence d'instructions exécutée après le traitant est celle qui suit la déclaration du traitant et non pas celle qui suit l'appel du signalant (paragraphe 2.1). Une première idée consiste à insérer un test après l'appel de la procédure signalante dans le corps de l'appelant. Nous ne la retenons pas car elle supprime la structuration offerte par un MTE. Il nous faut donc envisager de passer le "reste du programme" en paramètre. De plus, comme nous sommes en présence d'appels de procédures, il faut systématiquement appeler la procédure "reste du programme" pour n'exécuter la même

opération qu'une seule fois. Nous appelons cette procédure, la *procédure continuation* (sans vouloir anticiper, ce terme est emprunté à ce que nous présentons dans le chapitre suivant). Nous exposons cette proposition en détail. Tout appel à une procédure, qu'il s'agisse d'une procédure signalante ou non, nécessite de passer la procédure continuation en paramètre. Le corps de cette procédure correspond au code que l'on aurait écrit immédiatement après l'appel. Dans le cas d'un appel d'une procédure signalante, il est aussi nécessaire de passer autant de continuations qu'il y a d'exceptions pouvant être signalées par cette procédure. Une continuation pour une exception comprend le corps du traitant pour cette exception suivi de la séquence d'instructions figurant après le traitant. Dans l'exemple 2.2, la procédure B admettrait en paramètre deux continuations. La continuation standard serait $A2; A3$, et la continuation exceptionnelle liée à e serait $T; A3$. Nous présentons un exemple.

Exemple 2.10 : Nous donnons la version avec continuation de l'exemple 2.4. Nous pouvons remarquer que la continuation standard après appel du signalant et la continuation exceptionnelle sont identiques au traitant de l'exception près car l'appel du signalant *Convert* précède immédiatement la déclaration du traitant.

La continuation associée au reste du programme chez l'appelant de la procédure *Convert*, notée *cont_rest*, et la continuation associée au traitant de *code_incorrect*, notée *cont_code_incorrect*, sont de la forme suivante :

```
cont_rest = print ;  <reste du programme>
cont_code_incorrect = return ""; print ; <reste du programme>
```

Nous pouvons donner le corps de l'appelant :

```
Appelant =
   <sequence d'instructions>
   init_chaine;
   Convert(cont_rest, cont_code_incorrect);
```

Le corps de *Convert* consiste en un simple appel de procédure :

```
PROC Convert (PROC cont_rest, cont_car_inexistant) =
    INT i := 1;
    repr(code(i), cont_rest', cont_car_inexistant)
```

La continuation *cont_car_inexistant* est identique à *cont_code_incorrect* puisqu'il s'agit d'abandonner l'exécution de *Convert* lors du signal de *car_inexistant*. La continuation *cont_rest'* doit notamment permettre de traiter le reste du tableau. Nous donnons un aperçu de son contenu.

```
cont_rest' =
   IF fin_tableau
      THEN
        cont_rest
      ELSE
        i:=i+1;
        repr(code(i), cont_rest', cont_car_inexistant)
   ENDIF
```

Nous donnons la procédure *Repr* :

```
PROC Repr (INT ent ; PROC cont_rest', cont_car_inexistant) =
   IF non_representable(ent)
      THEN
        cont_car_inexistant
      ELSE
        rendre_convert_entier(ent);
        concat;
        cont_rest'
   ENDIF
```

◇

Cette approche est correcte au regard de ce que nous avons exigé pour un MTE puisque les programmes prenant en compte les exceptions sont structurés. Mais, le nombre de paramètres peut être facilement important puisqu'il y a autant de paramètres que d'exceptions pouvant être signalées. Cette solution est voisine de la définition dénotationnelle d'un MTE. Nous présentons cet aspect dans le chapitre 3. Nous pouvons aussi remarquer que la méthode proposée est plus satisfaisante que la définition d'un MTE pour un langage fonctionnel car elle ne fait pas appel à une facilité impérative. Néanmoins, l'utilisation de continuations pour la définition d'un MTE dans les langages "fonctionnels" n'est pas systématiquement retenue [Cous88] [Miln84]. Ceci est dû au fait que la méthode est considérée comme trop générale [Bret88]. Enfin, nous pouvons faire remarquer qu'un MTE fondé sur une approche statique a été proposé [Knud87]. Néanmoins, la concept de procédure est étendu de manière à permettre un transfert du contrôle après l'appel de la procédure traitante. Ceci évite de passer les continuations "reste du programme" en paramètre. De ces remarques, nous pouvons conclure qu'un MTE est nécessaire dans un langage impératif si l'on veut pouvoir écrire des applications réalistes structurées. Dans le paragraphe suivant nous présentons une évaluation des MTE.

2.4 Evaluation des MTE

Nous donnons tout d'abord des critères d'évaluation pour un MTE. Ensuite, nous comparons les modèles terminaison et RTR. Enfin, nous examinons les propagations implicite et explicite.

2.4.1 Critères d'évaluation d'un MTE

Nous énumérons quatre critères permettant l'évaluation d'un MTE [Cris79], la correction, l'uniformité, la simplicité, et l'économie.

La correction est liée à la définition du MTE. En effet, il ne suffit pas de définir un MTE en introduisant un certain nombre de primitives dans le langage, encore faut-il donner sa définition sémantique. Cette définition sémantique permet d'assurer que le MTE correspond à sa définition informelle, et de prouver la correction des programmes contenant des exceptions. Ce critère nous semble être le plus important car si la correction du MTE n'est pas assurée, il est impossible d'évaluer les critères suivants.

Le MTE doit être le même qu'il s'agisse d'une exception utilisateur, d'une exception détectée par le système ou d'une exception prédéfinie du langage. Nous appelons cet aspect, l'uniformité.

Le MTE doit être simple. Il ne doit pas introduire trop de primitives dans le langage. Et, la définition tant formelle qu'informelle du MTE doit être facilement compréhensible.

Enfin, l'économie se calcule par rapport au coût du MTE sur les temps d'implémentation, d'exécution et de maintenance.

2.4.2 Modèle RTR ou Terminaison ?

Si nous considérons le critère de simplicité d'un MTE, il apparait que le modèle terminaison est plus satisfaisant. En effet, si la syntaxe du mécanisme remplacement est réduite, ce MTE est plus difficile à comprendre qu'un MTE de type terminaison. Le résultat de la comparaison de ces deux modèles en terme d'économie est plus réservé. Le schéma reprise du modèle RTR permet de reprendre l'exécution d'une opération signalante alors que le modèle terminaison oblige une nouvelle exécution de l'opération. Mais, le schéma abandon du modèle RTR entraine un retour à l'opération signalante, action inexistante avec un modèle terminaison. Le choix en faveur de l'un ou de l'autre des modèles est lié à ce que l'on attend du MTE, c'est pourquoi nous ne faisons aucun choix a priori.

2.4.3 Propagation Explicite ou Implicite ?

La propagation implicite des exceptions rend plus difficile la vérification statique de l'existence des traitants et des exceptions dans le programme. Il faut donc, comme il est suggéré dans [Cris84] systématiquement vérifier que le programme est robuste. On est alors assuré que le programme termine à un point de sortie déclaré pour tout état d'entrée possible. A l'inverse, la propagation explicite

facilite la vérification statique de l'existence des traitants. Par ailleurs, la propagation implicite des exceptions peut avoir des conséquences dangereuses dans le cadre de la programmation modulaire [Cris87]. Supposons la hiérarchie de modules suivante : Les opérations du module N appellent des opérations du module M qui elles-mêmes appellent des opérations du module L. D'après le principe d'abstraction, le module N n'est pas supposé avoir connaissance des opérations de L utilisées par M. De plus, la propagation d'une exception signalée dans une opération O de L vers une opération Q de N viole le principe de base qu'après tout appel à O dans une procédure P de M, le contrôle doit retourner dans cette même procédure P. Enfin, cette propagation peut avoir pour effet de laisser le module M dans un état inconsistant. En conclusion, nous pensons que la propagation des exceptions doit être explicite.

Après cette présentation informelle du traitement d'exceptions, nous nous proposons d'étudier la sémantique des MTE dans les langages de programmation.

3 Sémantique dénotationnelle : Définition formelle des mécanismes de traitement d'exceptions

Ce chapitre est composé de deux parties. Nous introduisons tout d'abord la sémantique dénotationnelle en général, puis nous présentons la définition dénotationnelle des MTE.

3.1 Sémantique Dénotationnelle

Dans un premier temps, nous introduisons la sémantique dénotationnelle au moyen d'un exemple simple. Ensuite, nous présentons la sémantique dénotationnelle avec continuation. Ce dernier concept est nécessaire pour définir un MTE.

3.1.1 Sémantique dénotationnelle directe

La sémantique dénotationnelle permet de donner la sémantique d'un programme en terme de fonctions mathématiques définies sur des domaines particuliers [Schm86]. Nous choisissons un langage de programmation simple dont nous donnons la syntaxe:

P	::=	Function I (I_1 , ..., I_n) : I_{n+1}; C	*Fonction*
C	::=	C_1; C_2| if B then C_1 else C_2 end |I := E	*Commandes*
E	::=	E_1 + E_2| I | N	*Expressions*
B	::=	E_1 + E_2| non B	*Expressions booléennes*
I	::=	Identificateurs	*Identificateurs*
N	::=	Nombres	*Nombres*

Dans la déclaration de la fonction, I désigne le nom de la fonction, I_i où $i \in [1, n]$ désigne un paramètre, I_{n+1} désigne le résultat, et C désigne le texte.

La sémantique dénotationnelle d'une fonction est une fonction du domaine D^n (D contient les nombres) dans $D_\perp$. $D_\perp$ est défini comme étant $D \cup \perp$, où $\perp$ désigne les résultats indéfinis. Pour définir les domaines associées aux commandes, il nous faut considérer l'affectation. Il est nécessaire de pouvoir accéder et modifier un élément de la mémoire à partir d'un identificateur. Par conséquent, le domaine de la fonction sémantique commande est une fonction, appelée *Mémoire*, qui associe un nombre à un identificateur. Le co-domaine de la sémantique dénotationnelle d'une commande est aussi *Mémoire* car l'exécution d'une commande consiste à produire un nouvel état mémoire à partir de l'ancien. Les expressions ne font que consulter les données de la mémoire (I figure dans la définition de E) aussi, la sémantique dénotationnelle d'une expression est une fonction de *Mémoire* dans D. De la même manière, la fonction associée aux expressions booléennes est une fonction de *Mémoire* dans *Booléen*. Nous donnons les opérations du domaine *Mémoire*, puis les fonctions sémantiques du langage :

Domaine Mémoire = Id $\rightarrow$ D

Opérations
créermem : Mémoire
créermem = λi.$\perp$

accès : Id $\rightarrow$ Mémoire $\rightarrow$ D
accès = λi.λm.m(i)

maj : Id $\rightarrow$ D $\rightarrow$ Mémoire $\rightarrow$ Mémoire
maj = λi.λn.λm.[i $\rightarrow$ n] m

La fonction *créermem* permet de créer une nouvelle mémoire. La fonction *accès* permet d'accéder à la valeur associée à un identificateur dans une mémoire. La fonction *maj* permet de modifier la valeur associée à un identificateur dans une mémoire. La mémoire obtenue après exécution de cette fonction est identique à l'argument m excepté qu'elle associe la valeur n à l'identificateur i. [i $\rightarrow$ n] est une notation pour λid.id=i $\rightarrow$ n; m(id). Nous ne donnons pas les définitions des autres domaines, car elles s'obtiennent aisément.

La notation utilisée pour une fonction d'évaluation est le nom du domaine syntaxique de son argument en caractères gras. Par exemple, pour les commandes, le domaine syntaxique est C et on note la fonction d'évaluation C. *let* $x = e_1$ *in* e_2 indique que e_1 est associée à x dans e_2.

$\mathbf{F}$: Fonction $\rightarrow D^n \rightarrow D_\perp$
$\mathbf{F}$[Function I (I_1 , ..., I_n) : I_{n+1}; C] =
λ (N_1, ..., N_n).
let m_1 = (*maj* [I_1] N_1 *créermem*) in
...
let m_n = (*maj* [I_n] N_n m_{n-1}) in
let m' = $\mathbf{C}$[C] m_n in (*accès* [I_{n+1}] m')

$\mathbf{C}$: Commande $\rightarrow$ Mémoire $\rightarrow$ Mémoire
$\mathbf{C}[C_1; C_2] = \lambda$ m.$\mathbf{C}[C_2]$ ($\mathbf{C}[C_1]$ m)
$\mathbf{C}$[if B then C_1 else C_2 end] = λ m.$\mathbf{B}$ [B] m $\rightarrow$ $\mathbf{C}[C_1]$ m ; $\mathbf{C}[C_2]$ m
$\mathbf{C}[I := E] = \lambda$m.*maj* [I] ($\mathbf{E}$[E] m) m

$\mathbf{E}$: Expression $\rightarrow$ Mémoire $\rightarrow$ D
$\mathbf{E}[E_1 + E_2] = \lambda$m. $\mathbf{E}[E_1]$ m *plus* $\mathbf{E}[E_2]$ m
$\mathbf{E}[I] = \lambda$m. *accès* [I] m
$\mathbf{E}[N] = \lambda$m.$\mathbf{N}$[N]

$\mathbf{B}$: Expression booléenne $\rightarrow$ Mémoire $\rightarrow$ Booléen
$\mathbf{B}[E_1 = E_2] = \lambda$m. $\mathbf{E}[E_1]$ m *égale* $\mathbf{E}[E_2]$ m
$\mathbf{B}$[non B] = λm. *non*($\mathbf{B}$[B] m)

$\mathbf{N}$ est une fonction rendant la valeur d'un nombre.

Nous illustrons cette sémantique au moyen d'un exemple simple.

Exemple 3.1 : Il s'agit d'une fonction F qui rend la valeur 0 si son argument est égal à 10 et 1 sinon.

```
Function F(x) : res; if x = 10 then res := 0 else res := 1 end
```

Etant données les fonctions sémantiques définies précédemment, la fonction F est associée à la fonction sémantique suivante :

$\lambda(N_1)$ =
let m_1 = *maj* [x] N_1 *créermem* in
let m' =
((*accès* [x] m_1 *égale* $\mathbf{N}$[10] m_1) $\rightarrow$
maj [res] (0 m_1);
maj [res] (1 m_1))
in
(*accès* [res] m')

qui se simplifie en $\lambda(N_1) = N_1$ *égale* $10 \rightarrow 0; 1. \diamond$

Lors du signal d'une exception, il faut exécuter le traitant au lieu de la séquence d'instructions qui suit l'instruction de signal. Il est, par conséquent, nécessaire de pouvoir manipuler explicitement le contrôle. L'argument sémantique qui permet de modéliser le contrôle est la continuation. Les continuations ont initialement été introduites pour la modélisation des branchements absolus (instruction GOTO) et elles se sont avérées utiles pour la description de tout ordonnancement non standard. Dans la partie suivante, nous faisons une brève présentation de la sémantique dénotationnelle avec continuation.

3.1.2 Sémantique dénotationnelle avec continuation

Nous présentons un exemple simple [Schm86] qui nécessite un argument de contrôle. Nous considérons le langage du paragraphe 3.1.1 auquel nous ajoutons la commande *stop*. L'évaluation de cette dernière commande provoque le branchement à la fin du programme et la mémoire résultat est celle qui est fournie en argument. Une méthode pour modéliser le contrôle consiste à avoir une pile de contrôle comme argument de la fonction sémantique. La pile de contrôle maintient la liste de toutes les commandes à évaluer. La fonction d'évaluation dépile la commande du sommet de pile puis l'exécute. L'exécution du programme termine lorsque la pile est vide ou lorsque la commande *stop* est dépilée. Nous donnons un extrait de la fonction d'évaluation des commandes :

$\mathbf{C}$: commande $\rightarrow$ Pile_contrôle $\rightarrow$ Mémoire $\rightarrow$ Mémoire$_\perp$
$\mathbf{C}[C_1; C_2] = \lambda c.\lambda m.\mathbf{C}[C_1]\ (\mathbf{C}[C_2]\ \mathit{cons}\ c)\ m$
$\mathbf{C}[I := E] = \lambda c.\lambda m.\ (\mathit{hd}\ c)\ (\mathit{tl}\ c)\ (\mathit{maj}\ [I]\ (\mathbf{E}[E]\ m)\ m)$
$\mathbf{C}[\text{stop}] = \lambda c.\lambda m.\ m$
$\mathbf{C}[\text{if B then } C_1 \text{ else } C_2 \text{ end}] = \lambda c.\lambda m.\ \mathbf{B}[B]\ m \rightarrow (\mathbf{C}[C_1]\ c\ m); (\mathbf{C}[C_2]\ c\ m)$

Dans l'expression ($\mathbf{C}$[C] c m), ($\mathbf{C}$[C]) désigne l'expression de contrôle évaluée, *c* désigne la pile de contrôle, et *m* désigne l'argument mémoire. La pile est gérée comme une liste; les opérations *hd, tl* et *cons* désignent les opérations classiques sur les listes. Lors de l'évaluation de la composition séquentielle, l'expression de contrôle à évaluer est $\mathbf{C}[C_1]$, et la pile de contrôle devient la concaténation de $\mathbf{C}[C_2]$ avec la pile initiale (c'est à dire c). En ce qui concerne l'évaluation de l'affectation, l'expression de contrôle à évaluer étant une expression, on dépile la nouvelle commande à évaluer de la pile de contrôle. La commande *stop* désignant la terminaison du programme, la mémoire (c'est à dire m) est le résultat de l'exécution. Enfin, lors de l'évaluation de la commande conditionnelle, l'expression de contrôle à évaluer sera $\mathbf{C}[C_1]$ ou $\mathbf{C}[C_2]$ selon le résultat de l'évaluation de $\mathbf{B}$[B]. La pile peut être remplacée par une fonction qui est la continuation de commande. Son domaine est :

$$c \in \mathit{ContCmd} = \mathit{Mémoire} \rightarrow \mathit{Mémoire}_\perp.$$

Nous pouvons donner une nouvelle version de la fonction d'évaluation :

C : Commande → ContCmd → ContCmd
C[C1; C2] = λc. C[C1] (C[C2] c)
C[I:=E] = λc.λm. c (*maj*[I] (**E**[E] m) m)
C[stop] = λc.λm. m
C[if B then C1 else C2] = λc.λm. **B**[B] m → (C[C1] c m); (C[C2] c m)

L'argument c de continuation représente le *reste du programme* dans chacune des clauses. La continuation initiale étant exécutée à la fin du programme, elle peut contenir des instructions de "nettoyage" produisant le résultat final. Par exemple, si r est l'identificateur du résultat, la continuation initiale peut être $\lambda c.\lambda m.m(r)$. La forme générale du domaine d'une continuation de commande est :

$$c \in ContCmd = Mémoire \rightarrow Réponse$$

où *Réponse* peut être le domaine de la mémoire, des messages ou de toute autre fonction.

3.2 Définition dénotationnelle de deux MTE

Avant de donner la définition dénotationnelle de deux MTE, il nous faut présenter la méthode générale. C'est l'objet du paragraphe suivant.

3.2.1 Sémantique dénotationnelle du traitement d'exceptions

Nous devons tout d'abord présenter la définition dénotationnelle du concept de procédure. La présence de déclarations dans une procédure, ou dans un bloc, nous oblige à introduire la notion de contexte. Dans la sémantique dénotationnelle, le contexte est représenté par une valeur appelée environnement. Nous donnons les opérations de la fonction *Environnement* correspondante. Nous modifions aussi *Mémoire* pour avoir une gestion plus réaliste de la mémoire. La valeur d'un identificateur se retrouve à partir de l'adresse mémoire de celui-ci. Par conséquent, *Mémoire* devient une fonction de *Adresse* dans *Nat*. Le domaine *Nat* comprend les nombres. Nous présentons tout d'abord la définition de *Adresse :*

Domaine Adresse

Opérations
première-adr : Adresse
suivante-adr : Adresse → Adresse
égale-adr : Adresse → Adresse → Booléen
inférieure-adr : Adresse → Adresse → Booléen

Nous ne donnons que les domaines des opérations de *Adresse* car leur signification se déduit facilement. Nous présentons la nouvelle définition de *Mémoire* :

Domaine Mémoire = Adresse → Nat

Opérations
accès : Adresse → Mémoire → Nat
accès = λa.λm.m(a)

maj : Adresse → Nat → Mémoire → Mémoire
maj = λa.λn.λm.[a → n]m

Un élément de *Environnement* est un couple. Le premier élément est une fonction qui associe sa valeur à un identificateur. Le deuxième élément est une valeur d'adresse. L'environnement permet d'associer les valeurs de D à l'adresse mémoire la plus grande.

Domaine Environnement = (Id → D) x *Adresse*

Opérations
envvide : Adresse → Environnement
envvide = λa.((λi. in *Valerreur*()), a)

accèsenv : Id → Environnement → D
accèsenv = λi.λ(map, a).map(i)

majenv : Id → D → Environnement → Environnement
majenv = λi.λd.λ(map, a). ([i→d] map, a)

reserve-adr : Environnement → (Adresse x Environnement)
reserve-adr = λ(map, a). (a, (map, *suivante-adr*(a)))

Nous pouvons introduire les déclarations de variables et de procédures ainsi que l'appel de procédures dans le langage du paragraphe 3.1.1. Il faut définir le domaine syntaxique D (*Déclarations*) qui permet de déclarer les procédures et les variables. Le domaine syntaxique C (*Commandes*) doit être complété par l'appel de procédure. Pour simplifier, nous ne supposons que des variables de type entier :

D	::=	D_1; D_2 | var I | proc I= C	*Déclarations*
C	::=	... | begin D; C end | I	*Commandes*

Dans la définition des commandes, I désigne l'appel de procédure.

Nous proposons les fonctions d'évaluation du nouveau langage. Nous nous limi-

tons à la définition sémantique des déclarations et des commandes. Il faut noter que le domaine D est à présent la somme du domaines des nombres, des adresses et des procédures. Le domaine sémantique des procédures est :

$$Proc = ContCmd \rightarrow ContCmd$$

Nous donnons tout d'abord quelques définitions concernant la somme (ou union disjointe) de deux domaines. Les opérations qui permettent d'ajouter des éléments au domaine $R + S$ sont in$R$ et in$S$. L'opération in$R$ de R dans $R + S$ prend un élément r de R et lui adjoint un index de façon à indiquer qu'il provient de R. in$R(r)$ est donc égal à (*zero*, r). L'opération in$S$ est défini de la même manière sur S mais l'index est égal à *un*. L'opération Est$R$ (resp. Est$S$) de $R + S$ dans *Booléen* rend vrai si l'index de l'élément de $R + S$ est égal à *zero* (resp. *un*). La généralisation à la somme de plusieurs domaines est évidente.

D : Declaration → Environnement → Environnement
D[D1; D2] = **D**[D_2] o **D**[D_1]
D[var I] = λe.let(l', e') = (*reserve-adr* e) in (*majenv* [I] in*Adresse*(l') e')
D[proc I= C] = λe. (*majenv* [I] in*Proc*(**C**[C] e) e)

C : Commande → Environnement → ContCmd → ContCmd
C[C_1; C_2] = λe. λc.**C**[C_1]e (**C**[C_2]e c)
C[I := E] = λe.λc.λm. c (*maj* (*accèsenv* [I]) (**E**[E]e m) m)
C[if B then C_1 else C_2 end] = λe.λc. λm. **B**[B]e m → (**C**[C_1]e c m); (**C**[C_2]e c m)
C[begin D; C end] = λe.λc. **C**[C] (**D**[D]e c) c
C[I] = λe.λc.cas (*accèsenv* [I] e) : Est*Proc*(q) → (q c) end

E : Expression → Environnement → Mémoire → Nat
(Non détaillé)

B : Expression booléenne → Environnement → Mémoire → Booléen
(Non détaillé)

Nous sommes en mesure d'aborder la définition dénotationnelle des MTE. Le problème n'est pas neuf puisqu'il est proche de ce que nous avons exposé dans le paragraphe 2.3.2. L'idée consiste à ajouter les continuations, associées à chaque traitant, dans l'environnement du signalant lors de l'évaluation de la commande d'appel d'un signalant. Dans le cas d'un modèle terminaison, la continuation du traitant comprend le corps du traitant suivie de la séquence d'instructions qui se trouve après la liste de déclaration des traitants. Lors du signal d'une exception, on recherche dans l'environnement, la continuation pour le traitant de l'exception. Ensuite, la continuation courante est écartée et l'exécution continue avec la continuation extraite de l'environnement. Nous donnons succintement la définition dénotationnelle d'un MTE de type terminaison.

3.2.2 Définition dénotationnelle d'un MTE de type terminaison

Le MTE que nous définissons possède une syntaxe identique au MTE du langage MODULA-2 (paragraphe 2.2.2). Pour simplifier les fonctions sémantiques, les exceptions signalées par une procédure sont déclarées dans l'en-tête de celle-ci. Ceci nous permet de différencier les procédures signalantes des autres et donc de savoir s'il faut modifier l'environnement de la procédure lors de l'évaluation de l'appel. Nous ne vérifions pas l'existence systématique des déclarations d'exceptions afin de ne pas nous soucier de la sémantique statique des programmes. Toujours dans un souci de simplicité, nous interdisons le passage de paramètres aux traitants. Nous donnons la syntaxe du MTE :

D	::=	... | proc I_1 raises I_2 ... I_n = C	*Déclarations*
C	::=	... | try C except T end | raise I	*Commandes*
T	::=	T1; T2 | exc : C	*Déclaration des traitants*

proc I_1 raises I_2 ... I_n = C permet de déclarer une procédure signalante. I_k, $k \in [2,n]$ désigne une exception. Enfin, il faut noter que le signalant peut être une procédure où toute autre commande du langage.

Nous pouvons suggérer les fonctions d'évaluation du MTE. Le domaine D peut contenir des continuations (*ContCmd*) en plus de ce que nous avons précédemment cité. Nous rappelons ou introduisons certains domaines.

Mémoire	=	Adresse → Nat
Proc	=	ContCmd → ContCmd
ProcSignalante	=	Environnement → ContCmd → ContCmd
ContCmd	=	Mémoire → Mémoire
D	=	Nat + Adresse + Proc + ProcSignalante + ContCmd

D : Déclaration → Environnement → Environnement
D[proc I_1 raises I_2, ..., I_n = C] = λe. (*majenv* [I_1]
in*ProcSignalante* (λe'. **C**[C]
(*majenv* [I_n] (*accèsenv* [I_n] e')
...
(*majenv* [I_3] (*accèsenv* [I_3] e')
(*majenv* [I_2] (*accèsenv* [I_2] e')) e))...)
e)

C : Commande → Environnement → ContCmd → ContCmd
C[I] = λe.λc.cas (*accèsenv* [I] e) :
Est*Proc*(q) → (q c) |
Est*ProcSignalante* (r) → (r e c)
end

$\mathbf{C}$[try C except T end] = λe.λc. $\mathbf{C}$[C] ($\mathbf{T}$[T] e e c) c
$\mathbf{C}$[raise I] = λe.λc. cas (*accèsenv* [I] e) : Est*ContCmd*(c') $\rightarrow$ c' end

$\mathbf{T}$: Declaration_traitant $\rightarrow$ Environnement $\rightarrow$ Environnement
$\rightarrow$ ContCmd $\rightarrow$ Environnement
$\mathbf{T}$[T1;T2] = λe.λe'.λc.$\mathbf{T}$[T2] ($\mathbf{T}$[T1] e e' c) e' c
$\mathbf{T}$[exc:C] = λe.λe'.λc. *majenv*[exc] in*ContCmd*($\mathbf{C}$[C] e' c) e

Dans l'évaluation de la déclaration d'une procédure signalante, e' désigne l'environnement de l'appelant. La mise à jour de l'environnement de la procédure (c'est à dire e) permet d'y inclure la continuation du traitant associé à l'exception. Dans le cas où le signalant est une commande du langage, la continuation du traitant appartient à l'environnement aussi il n'est pas nécessaire d'introduire des définitions particulières.

Il est important de noter que dans la définition sémantique de la déclaration d'un traitant, la continuation affectée au traitant recouvre la continuation pour le reste du programme. En effet, cette continuation est obtenue à partir de$\mathbf{C}$[C] qui correspond au corps du traitant et de c qui est la continuation pour le reste du programme. Par ailleurs, l'environnement de la continuation d'un traitant est l'environnement sans les traitants de la même clause de déclaration. Ceci est nécessaire si l'on veut pouvoir définir les mêmes identificateurs d'exceptions dans des niveaux différents (les déclarations de traitants peuvent être statiquement imbriquées). Nous donnons un exemple pour illustrer cette sémantique.

Exemple 3.2 : Nous reprenons l'exemple 2.4. Pour ne pas avoir à introduire la définition du plus petit point fixe, nous n'ajoutons pas la clause itérative au langage. Cet aspect de la sémantique dénotationnelle n'est pas important pour ce que nous présentons. Nous modifions quelque peu notre exemple. La procédure *Convert* qui consistait à convertir un tableau d'entiers en une chaîne de caractères, consiste à présent à convertir deux entiers en deux caractères. Nous donnons le corps des procédures *Convert* et *Repr*. Nous rappelons que la procédure *Repr* convertit un entier en un caractère, et qu'elle peut éventuellement signaler *car_inexistant*. Par ailleurs, comme nous n'avons pas introduit la définition des paramètres, ces procédures consultent et modifient des variables globales.

```
proc Appelant_de_convert =

var i  : INTEGER;
var n1 : INTEGER;
var n2 : INTEGER;
var c1 : CAR;
var c2 : CAR;
```

```
proc Convert =
   try
     i:=1; Repr; i:=2; Repr
   except
     car_inexistant : c1:=''; c2 :=''
   end
end

proc Repr raises car_inexistant =
   if i=1 then
      if min_ent_repr - 1 < n1 < max_ent_repr + 1
         then c1 := evalcar(n1)
         else raise car_inexistant
      end;
   else
      if min_ent_repr - 1 < n2 < max_ent_repr + 1
         then c2 := evalcar(n2)
         else raise car_inexistant
      end;
   end
end

lire(n1);
lire(n2);
convert;
end
```

Nous notons le caractère "vide" ϵ. Nous définissons les autres notations employées:

$$
\begin{array}{lcl}
C_1 & = & \text{i:=1; Repr; i:=2; Repr} \\
T & = & \text{car\_inexistant : c1:= } \epsilon\text{; c2:= } \epsilon \\
C_2 & = & \text{i:=2; Repr} \\
C_3 & = & \text{if } B_1 \text{ then } C_5 \text{ else raise car\_inexistant end} \\
C_4 & = & \text{if } B_2 \text{ then } C_6 \text{ else raise car\_inexistant end} \\
B_1 & = & \text{min\_ent\_repr} - 1 < \text{n1} < \text{max\_ent\_repr} + 1 \\
B_2 & = & \text{min\_ent\_repr} - 1 < \text{n2} < \text{max\_ent\_repr} + 1 \\
C_5 & = & \text{c1} := \text{evalcar(n1)} \\
C_6 & = & \text{c2} := \text{evalcar(n2)}
\end{array}
$$

Nous donnons l'évaluation de la procédure *Appelant_de_convert* à partir de l'appel de la procédure *Convert*. Nous supposons en outre que $n1$ n'est pas représentable.

Pour continuation initiale, nous prenons la continuation *fin*. Cette continuation vaut $\lambda m.m$ et permet de restituer la mémoire à la fin de l'exécution du pro-

gramme. L'environnement et la mémoire valent respectivement e_0 et m_0 lors de l'appel de la procédure *Convert*. Les variables étant des variables globales, elles font parties des environnements des procédures.

e_1 désignant l'environnement de la procédure, nous obtenons :

C[Convert] e_0 fin m_0 =
C[try C_1 except T] e_1 fin m_0 =
C[C_1] (**T**[T] e_1 e_1 fin) fin m_0 =
C[C_1] (*majenv* [car_inexistant] in*ContCmd*(**C**[c1:=ϵ; c2:=ϵ] e_1 fin) e_1) fin m_0

Nous posons e_2 = *majenv* [car_inexistant] in*ContCmd*(**C**[c1:=ϵ; c2:=ϵ] e_1 fin) e_1. Par conséquent :

C[C_1] (*majenv* [car_inexistant] in*ContCmd*(**C**[c1:=ϵ; c2:=ϵ] e_1 fin) e_1) fin m_0 =
C[i:=1; Repr; C_2] e_2 fin m_0 =
C[i:=1] e_2 (**C**[Repr; C_2] e_2 fin) m_0 =
C[Repr; C_2] e_2 fin (*maj* (*accèsenv* [i]) (**E**[1] e_2 m_0) m_0)

Soit m_1 = *maj* (*accèsenv* [i]) **E**[1] e_2 m_0), nous obtenons :

C[Repr; C_2] e_2 fin (*maj* (*accèsenv* [i]) (**E**[1] e_2 m_0) m_0) =
C[Repr; C_2] e_2 fin m_1 =
C[Repr] e_2 (**C**[C_2] e_2 fin) m_1

Le domaine de Repr est *ProcSignalante* et sa valeur est :
λe'.**C**[if i=1 then C_3 else C_4end] (*majenv* [car_inexistant] (*accèsenv* [car_inexistant] e') e_3). On obtient :

C[Repr] e_2 (**C**[C_2] e_2 fin) m_1 =
C[if i=1 then C_3 else C_4 end]
(*majenv* [car_inexistant] (in*ContCmd*(**C**[c1:=ϵ;c2:=ϵ] e_1 fin) e_3)
(**C**[C_2] e_2 fin) m_1

Puisque e_2 = *majenv* [car_inexistant] in*ContCmd*(**C**[c1:=ϵ; c2:=ϵ] e_1 fin) e_1. Nous posons e_4 = (*majenv* [car_inexistant] (in*ContCmd*(**C**[c1:=ϵ;c2:=ϵ] e_1 fin) e_3). Sachant, enfin, que e_3 désigne l'environnement de la procédure, on a :

C[if i=1 then C_3 else C_4 end]
(*majenv* [car_inexistant] (in*ContCmd*(**C**[c1:=ϵ;c2:=ϵ] e_1 fin) e_3)
(**C**[C_2] e_2 fin) m_1 =
C[if i=1 then C_3 else C_4end] e_4(**C**[C_2] e_2 fin) m_1 =
B[i=1] e_4 m_1 $\rightarrow$ (**C**[C_3] e_4 (**C**[C_2] e_2 fin) m_1) ; (**C**[C_4] e_4 (**C**[C_2] e_2 fin) m_1)

Nous ne présentons pas l'évaluation de l'expression booléenne (i=1) , le résultat de cette évaluation est égal à vrai. Par conséquent, il nous reste à évaluer :

C[if B_1 then C_5 else raise car_inexistant end] e_4 (C[C_2] e_2 fin) m_1 =
B[B_1] e_4 m_1 →
(C[C_5] e_4 (C[C_2] e_2 fin) m_1) ; (C[raise car_inexistant] e_4 (C[C_2] e_2 fin) m_1)

Nous ne présentons pas l'évaluation de l'expression booléenne B_1, Le résultat de cette évaluation est égal à faux. Nous devons donc évaluer :

C[raise car_inexistant] e_4 (C[C_2] e_2 fin) m_1 =
cas (*accèsenv* [car_inexistant] e_4) : Est*ContCmd*(C[c1:=ϵ; c2:ϵ] e_1 fin) →
C[c1:=ϵ; c2:=ϵ] e_1 fin m_1

Puisque e_4 = (*majenv* [car_inexistant] (in*ContCmd*(C[c1:=ϵ;c2:=ϵ] e_1 fin) e_3), nous obtenons :

cas (*accèsenv* [car_inexistant] e_4) : Est*ContCmd*(C[c1:=ϵ; c2:=ϵ] e_1 fin) →
C[c1:=ϵ; c2:=ϵ] e_1 fin m_1 =
C[c1 :=ϵ] e_1 (C[c2:=ϵ] e_1 fin) m_1

Nous ne présentons pas l'évaluation de la commande, la mémoire modifiée après cette évaluations est notée m_2. Par conséquent, nous obtenons :

C[c2:=ϵ] e_1 fin m_2

La mémoire modifiée après cette évaluation étant notée m_3, il nous reste à évaluer :

fin m_3 =
m_3

On termine donc bien dans un état où *c*1 et *c*2 ont pour valeur ϵ. ◇

Nous concluons ce chapitre en donnant succintement la définition dénotationnelle d'un MTE de type RTR.

3.2.3 Définition dénotationnelle d'un MTE de type RTR

Dans le cas d'un modèle RTR, on peut vouloir reprendre l'exécution de la continuation courante après exécution du traitant. Si nous reprenons le mécanisme remplacement (paragraphe 2.2.3), l'absence ou la présence du mot clé REPLACE à la fin du corps du traitant indique respectivement la reprise ou la terminaison du signalant. Par conséquent, en fonction de ce mot clé, on peut adjoindre la bonne continuation au corps du traitant. Dans le MTE que nous proposons, le

mot clé associé à la reprise du signalant est *resume*. La définition dénotationnelle du schéma terminaison du modèle RTR est différente de celle du modèle terminaison; il faut exécuter la séquence d'instructions qui suit l'appel du signalant après l'exécution du traitant. Nous ne présentons pas le schéma réessai car il peut se déduire facilement. La déclaration des traitants est donc la suivante :

T ::= T1; T2 | exc : C | exc : C; resume

Nous proposons la définition sémantique d'un tel mécanisme. La définition de **T**[T1;T2] est inchangée.

T : Déclaration_traitant → Environnement → Environnement
→ ContCmd → Environnement
T[T1; T2] = λe.λe'.λc.**T**[T2] (**T**[T1] e e' c) e' c
T[exc : C] = λe.λe'.λc. *majenv* [exc] (in*ContCmd* **C**[C] e') e
T[exc : C; resume] = λe.λe'.λc. *majenv* [exc] (in*Traitant* **C**[C] e') e

Il faut modifier la définition de la déclaration d'une procédure signalante et de la commande de signal. Le domaine D peut aussi contenir des traitants (*Traitant*) de type $ContCmd \rightarrow ContCmd$.

D : Déclaration → Environnement → Environnement
D[proc I_1 raises I_2, ..., I_n = C] = λe.(*majenv* [I_1]
in*ProcSignalante* (λe'.λc **C**[C]
(*majenv* [I_n] ((*accèsenv* [I_n] e') c)
...
(*majenv* [I_3] ((*accèsenv* [I_3] e') c)
(*majenv* [I_2] ((*accèsenv* [I_2] e') c) e))...)
e)

C : Commande → Environnement → ContCmd → ContCmd
C[raise I] = λe.λc.cases (*accèsenv* [I] e) of
Est*ContCmd*(c') → c' |
Est*Traitant*(t) → t c
end

Nous ne développons pas plus la définition dénotationnelle des MTE. Dans le chapitre suivant, nous introduisons la sémantique axiomatique des MTE.

4 Sémantique axiomatique : Vérification de programmes contenant des exceptions

Dans ce chapitre, nous nous intéressons plus à l'aspect preuve de programme qu'à la définition sémantique des MTE. L'approche suivie est similaire quelque soit la solution proposée. Les différents cas exceptionnels figurent dans la spécification, la postcondition est la disjonction de l'effet du cas standard et des effets du traitement des exceptions. La preuve de la correction du programme avec sa spécification consiste à vérifier que le programme signale les exceptions s'il y a occurence des conditions exceptionnelles et que les effets des traitants sont correctes au regard de la postcondition. Les travaux réalisés sur la sémantique axiomatique du traitement d'exceptions se décomposent en deux familles. Il y a d'une part les axiomes fondés sur la logique de Hoare [Cocc82][Levi77][Lukh80][Yemi87], et d'autre part les axiomes fondés sur l'utilisation de la formulation de la précondition la plus faible (*wp* pour "Weakest Precondition")[Cris84]. Ces deux familles font chacune l'objet d'un paragraphe.

4.1 Axiomes fondés sur la logique de HOARE

La logique de Hoare est une sémantique axiomatique où les axiomes et les règles d'inférence spécifient le comportement des instructions du langage. Une proposition de la forme $P\{C\}Q$ (P et Q sont des fonctions booléennes) signifie que si P est vraie avant l'évaluation de C, et si C termine alors Q sera vraie après l'évaluation de C. P est la précondition et Q est la postcondition. Nous donnons l'axiome pour la composition séquentielle.

$$\frac{P\{C_1\}Q,\ Q \Rightarrow R,\ R\{C_2\}S}{P\{C_1; C_2\}S}$$

Nous ne présentons en détail que deux définitions axiomatiques de MTE parmi celles existantes. Notre choix s'est porté sur le travail effectué pour le MTE du langage ADA car le système d'axiomes a été pensé pour un MTE existant. Nous étudions aussi la sémantique axiomatique du mécanisme remplacement introduit dans le paragraphe 2.2.3.

4.1.1 Sémantique axiomatique du MTE du langage ADA

Le MTE du langage ADA [Ada79] met en œuvre le modèle terminaison. La propagation des exceptions est implicite excepté dans le cas d'un processus. Une exception n'est propagée entre deux processus que s'ils communiquent de manière synchrone. L'exécution de la commande *raise t.failure* permet de signaler une exception d'un processus vers un autre, ici le processus t. Le traitant de l'exception *failure* dans le processus t permet en général d'effectuer des actions de "nettoyage" avant l'abandon du processus. Lorsque deux processus tentent de communiquer entre eux ou sont en train de le faire, l'occurence d'une exception chez l'un

d'eux peut avoir des repercussions chez l'autre par le biais de l'exception standard *tasking-error*. Une telle exception est signalée si la communication est impossible, si une exception est propagée de l'appelant vers l'appelé ou enfin, si l'un des deux processus est interrompu par un autre processus (par une instruction *abort* ou par une exception *failure*). Nous présentons la sémantique axiomatique donnée pour ce MTE [Lukh80]. Il est important de noter qu'il n'existe aucune règle pour spécifier l'exception *failure* des processus, et nous pouvons le regretter. Il semble que ce soit un problème particulièrement complexe. Il a fallu ajouter une documentation formelle aux exceptions pour pouvoir définir des axiomes. Cette documentation est déclarée au moyen d'instructions de la forme ASSERT qui sont des assertions logiques non exécutées. Les assertions logiques sont des instructions de spéfication plus générales que les conditions ADA mais elles obéissent aux règles de visibilité classiques. Toutes les variables libres sont les variables visibles dans le programme. Nous décrivons la documentation formelle du traitement d'exceptions. A chaque traitant, est associée une assertion qui doit être vraie lorsque l'exception qui lui est associée est signalée. Les exceptions propagées sont déclarées, et on associe à chacune d'elles une assertion qui doit être vraie lors de la propagation. Nous reprenons l'exemple 2.4.

Exemple 4.1 : Nous nous plaçons dans le cas où l'exception *car_incorrect* est propagée vers l'appelant de la procédure *Convert*. La fonction *repr_car* donne le caractère représentant l'entier passé en paramètre. *repr_car* est identique à *Repr* mais nous devons le nommer autrement pour la preuve. Une preuve complète devrait montrer que *Repr* rend effectivement le même résultat que *repr_car*. Nous le supposons. Nous donnons le corps de la procédure *Convert* :

```
TYPE tableau IS ARRAY[1..n] OF INTEGER
PROCEDURE Convert (code :IN tableau,s : OUT STRING)
ENTRY Vrai
EXIT
1 ≤ i ≤ n ∧ s[i] = repr_car(code[i])
PROPAGATE car_inexistant
ASSERT
∃i : 1 ≤ i ≤ n, code[i] > max_ent ∨ code[i] < min_ent
BEGIN
< Corps de Convert >
END
```

max_ent (resp. *min_ent*) désigne le plus grand (resp. le plus petit) entier représentable. Le mot clé ENTRY permet de spécifier la précondition, EXIT permet de spécifier la postcondition, PROPAGATE permet de déclarer les exceptions propagées, et ASSERT permet de spécifier une assertion. Dans notre exemple, il s'agit de l'assertion associée à l'exception *car_inexistant*. Nous présentons l'appelant de *Convert* :

ASSERT Vrai
BEGIN
Convert(code, s)
EXCEPTION
WHEN car_inexistant ASSERT
$\exists j : 1 \leq j \leq n,\ code[i] > max\_ent \vee code[i] < min\_ent \Rightarrow$ s:="" ;
END
ASSERT
$(\forall j : 1 \leq j \leq n,\ s[j] = repr\_car(code[j])) \vee$
$(\exists j : 1 \leq j \leq n, (code[i] > max\_ent \vee code[i] < min\_ent) \wedge s = "")$

Nous trouvons l'assertion associée au traitant de *car_inexistant* (1ère assertion dans le texte) dans la clause de déclaration des traitants, et la postcondition de l'appelant (2ème assertion). $\diamond$

Cette documentation permet d'assurer qu'une exception a toujours une assertion associée quelque soit le contexte où elle est signalée. On en déduit l'axiome de signal d'une exception.

A1 - Axiome de signal d'une exception :

$$A_i \ \{ \text{ raise } E_i \} Faux$$

où A_i est l'assertion associée à l'exception E_i , et *raise* l'instruction de signal. Il faut, par ailleurs, que tout traitant d'exception réalise l'assertion de sortie du bloc dans lequel il est. Cette assertion pouvant être supposée dans la clause "WHEN", on obtient la règle de preuve pour un bloc C.

R1 - Règle de preuve d'un bloc :

$$\frac{P\{S_0\}Q, A_1\{S_1\}Q, ..., A_n\{S_n\}Q}{P\{C\}Q}$$

où le corps de C est écrit comme suit :

BEGIN
S_0
EXCEPTION
WHEN E1 ASSERT
$A_1 \Rightarrow S_1$
...
WHEN En ASSERT
$A_n \Rightarrow S_n$

```
END
```

Nous donnons enfin la règle de preuve qui décrit l'effet d'une exception propagée par une procédure. Soit p, une procédure qui propage les exceptions $E_1, ..., E_n$. $A_1, ..., A_n$ sont les assertions associées aux exceptions dans la déclaration des propagations de la procédure. Soit B_j l'assertion associée à E_j dans l'environnement de l'appelant. B_j peut donc soit être une assertion associée au traitant de l'appelant, soit être une assertion de la déclaration des propagations de l'appelant. La règle de preuve de l'appel à p nécessite que pour toutes les exceptions E_i propagées, $A_i \Rightarrow B_i$. Ceci permet d'assurer la postcondition de l'appel à p. Les paramètres formels d'entrée, d'entrée-sortie et de sortie sont respectivement notés fi, fio et fo. On suppose que la correction du corps de p a été établie en fonction de la condition d'entrée $I(fi, fio)$ et de la condition de sortie $O(fi, fo, fio)$. Nous pouvons founir la règle de preuve pour un appel à p.

R2 - Règle de preuve d'un appel de procédure :

$$\frac{\begin{array}{c}(P \Rightarrow I(ri, rio)) \wedge \forall ro, rio(O(ri, ro, rio) \Rightarrow Q), \\ (P \Rightarrow I(ri, rio)) \wedge \forall ro, rio(A_1(ri, ro, rio) \Rightarrow B_1), \\ \ldots \\ (P \Rightarrow I(ri, rio)) \wedge \forall ro, rio(A_n(ri, ro, rio) \Rightarrow B_n),\end{array}}{P\{p(ri, ro, rio)\}Q}$$

où ri, rio et ro désignent respectivement les paramètres réels d'entrée, d'entrée-sortie et de sortie de la procédure. Nous donnons la preuve de l'exemple 4.1 afin d'illustrer cette axiomatique.

Preuve de la correction de l'exemple 4.1 : Etant données R1 et R2, cette preuve se décompose en trois étapes. Les étapes 1 et 2 sont nécessaires à la preuve de R2, et la troisième étape est nécessaire à la preuve de R1 :

- L'assertion de sortie de la procédure *Convert* doit impliquer l'assertion de sortie du bloc appelant.
- L'assertion associée à l'exception *car_inexistant* dans la procédure *Convert* doit impliquer l'assertion associée à cette même exception dans l'environnement appelant.
- Le traitant doit remplir la condition de sortie.

Preuve des prémisses de R2 :

Dans la mesure où la précondition du bloc appelant et la précondition de la procédure *Convert* sont vrais, $P \Rightarrow I(ri, rio)$ est vérifié.

$\forall ro, rio(O(ri, ro, rio) \Rightarrow Q)$ qui devient :

$\forall s((\forall i : 1 \leq i \leq n, s[i] = repr\_car(code[i]))$
$\Rightarrow (\forall j : 1 \leq j \leq n, s[j] = repr\_car(code[j]))\vee$
$((\exists j : 1 \leq j \leq n, code[j] > max\_ent \vee code[j] < min\_ent) \wedge s = ""))$

est vrai. Et, $\forall ro, rio(A_1(ri, ro, rio) \Rightarrow B_1)$ qui devient :

$\forall s(\exists i : 1 \leq i \leq n, ((code[i] > max\_ent) \vee (code[i] < min\_ent))$
$\Rightarrow \forall s(\exists j : 1 \leq j \leq n, ((code[j] > max\_ent) \vee (code[j]¡min\_ent))$

est vrai. Par conséquent Les prémisses de la règle R2 sont vérifiés. On obtient :

$$Vrai\{\texttt{Convert(code, s)}\}Q$$

Preuve des prémisses de R1 :

Nous venons de prouver le corps de l'appelant puisqu'il consiste en un simple appel à *Convert,* il nous reste à prouver :

$$B_1\{\texttt{s} := ""\}Q$$

qui est la troisième et dernière étape de la preuve.

$$((\exists j : 1 \leq j \leq n, ((code[j] > max\_ent) \vee (code[j] < min\_ent))$$
$$\{ (\text{s:=""}) \}$$
$$(\forall j : 1 \leq j \leq n, s[j] = repr\_car(code[j])) \vee$$
$$((\exists j : 1 \leq j \leq n, code[j] > max\_ent \vee code[j] < min\_ent) \wedge s = ""))$$

La postcondition étant vérifiée, nous pouvons conclure :

$$Vrai\{C\}Q$$

$\diamond$

La méthode proposée s'étend de manière naturelle aux modules. Il faudrait par contre un travail plus important pour donner une sémantique axiomatique de l'exception *failure.* La déclaration des propagations permet d'être plus précis dans la spécification des programmes ADA mais elle est contestable dans la mesure où elle restreint le MTE initial du langage. Il ne s'agit plus d'un MTE avec propagation implicite. Nous présentons la sémantique axiomatique du mécanisme remplacement.

4.1.2 Sémantique axiomatique du mécanisme remplacement

Nous faisons un bref rappel du mécanisme remplacement déjà introduit dans le paragraphe 2.2.3. Ce MTE est de type RTR. Si le traitant termine en exécutant l'instruction REPLACE alors le signalant est abandonné et la valeur de l'expression calculée par le traitant est retournée à l'appelant du signalant. Sinon, la valeur calculée par le traitant est retournée au signalant et l'exécution de ce dernier reprend immédiatement après le point de signal. Enfin, les traitants sont déclarés entre les mots clés ON et NO. L'axiomatisation du mécanisme remplacement [Yemi87] est dérivée de celle proposée pour le langage ALGOL 68 [Schw79]. Cette axiomatisation est bien adaptée au mécanisme remplacement car elle contient des règles d'inférence pour des procédures qui autorisent des paramètres de type arbitraire (par exemple les procédures). De plus, elle n'implique aucune restriction sur les effets de bord dans les expressions. Ce point est important car les effets de bord surviennent naturellement dans le traitement des exceptions. En effet, toute action exécutée par un traitant est un effet de bord de l'appel du signalant. Les phrases de la logique de [Schw79] sont toutes de la forme suivante:

$$N/P\{s\}Q \wedge \sigma = v$$

où
s désigne une expression ou une instruction,
σ désigne la valeur rendue par s,
P désigne l'assertion d'entrée,
$Q \wedge \sigma = v$ désigne l'assertion de sortie, et
N peut être assimilé à l'environnement de la sémantique dénotationnelle. Il permet entre autre de donner le type de tout identificateur, et par conséquent fournit les propriétés statiques nécessaires à la preuve.
$N/$ signifie que N est distribué sur toutes les parties de la phrase.

La définition axiomatique du MTE fait intervenir plusieurs règles de preuves. Il faut spécifier le signalant et ses exceptions, les effets d'un traitant, et la règle d'appel d'un signalant. Nous présentons ces règles.

La spécification d'un signalant et de ses exceptions se fait en considérant l'assertion de sortie du cas normal et les assertions de sortie des différents cas exceptionnels. Il faut, de plus, fournir une condition de reprise qui doit être vérifiée, lors de la reprise, après exécution du traitant. Cette condition est indispensable pour assurer l'assertion de sortie du cas normal quand s termine, s'il termine. Nous sommes en mesure de donner la spécification d'un signalant s qui signale une exception e. Cette exception admet pour paramètre formel, le vecteur x.

S1 - Spécification d'un signalant :

$$N/\{P, Q \wedge \sigma = v, e(x) < E(x), R(x) \wedge \sigma = u >\}$$

où
P désigne l'assertion d'entrée,
$Q \wedge \sigma = v$ désigne l'assertion de sortie du cas normal,
E désigne la condition d'exception associée à e, et
$R(x) \wedge \sigma = u$ désigne la condition de reprise.

On dit que s est partiellement correct étant donné cette spécification ou encore :

$$s\ PC\ wrt\ (N/\{P, Q \wedge \sigma = v, e(x) < E(x), R(x) \wedge \sigma = u >\})$$

si et seulement si on a :

P1 - Correction partielle :

$$\frac{\forall\ traitant\ h\ pour\ e,\ N/E(x)\{h(x)\}R(x) \wedge \sigma = u}{N/P\{s\}Q \wedge \sigma = u}$$

Dans le cas général, il y a plusieurs couples de la forme $\{P, Q \wedge \sigma = u\}$ avec plusieurs exceptions associées. Nous ne le présentons pas car l'extension est relativement simple.

Pour spécifier les effets d'un traitant, il suffit de donner une règle pour la primitive REPLACE. La reprise fait intervenir le même mécanisme que pour les appels de procédure, et la terminaison d'une construction englobant un appel est obtenue par une simple généralisation de la notion de signalant. Comme, après l'exécution de REPLACE, le contrôle va quelque part ailleurs (l'adresse de retour du signalant), tout peut être supposé. C'est pourquoi, la postcondition de REPLACE est *Faux*.

R1 - Règle de REPLACE :

$$\frac{N/P\{e\}Q \wedge \sigma = v}{N/P\{e\ REPLACE\}Faux \wedge (REPLACE : Q \wedge \sigma = v)}$$

où la notation "REPLACE :" est empruntée à la logique temporelle. Les assertions de la forme "Etiquette : Prédicat" spécifient que le prédicat doit être vrai à l'étiquette spécifiée dans le programme.

La règle d'appel d'un signalant combine des spécifications indépendantes; nous trouvons celle du signalant et celles des traitants désignés pour un appel particulier de ce signalant. Nous présentons la règle dans le cas où la procédure et le traitant sont des identificateurs, et la règle dans le cas où le signalant est une commande du langage.

R2 - Règle de l'appel simple :

$$\frac{\begin{array}{l}1.N/T \wedge INV\{COLLAT(e_1,...,e_n)\}P \wedge INV \wedge \sigma_e = x \\ 2.p(x)\ PC\ wrt\ (N/\{P, Q \wedge \sigma_0 = v, ex(z) < E(z), R(z) \wedge \sigma_1 = u >\}) \\ 3.N/E(z) \wedge INV\{h(z)\}(R(z) \wedge INV \wedge \sigma_1 = u) \vee (REPLACE : S \wedge INV \wedge \sigma_2 = w)\end{array}}{N/T \wedge INV\{p(e_1,...,e_n)\ ON\ ex = h\ NO\}(Q \vee S) \wedge INV \wedge \sigma = (Q|v) \oplus (S|w)}$$

Où
$(Q|v)$ doit être lu *si* Q *alors* v,
$(v_1 \oplus v_2)$ doit être lu v_1 *ou* v_2, et
$(p(x)\ PC\ wrt\ S)$ doit être lu *p(x) partiellement correct avec la spécification* S .

Avant d'analyser les prémisses de cette règle nous précisons la notation utilisée. T, P, S et Q désignent respectivement la précondition de l'appel, la précondition de la procédure appelée, la postcondition de l'appel dans le cas d'un remplacement et la postcondition classique de l'appel. INV est une assertion sur l'environnement de l'appelant. Ceci est nécessaire puisque le traitant et l'évaluation collatérale (COLLAT$(e_1,...,e_n)$) peuvent avoir des effets de bord sur l'environnement de l'apelant. L'environnement de l'appelant n'est pas accessible au signalant aussi il n'apparaît pas dans le prémisse 2.

Le prémisse 1 reflète les effets de l'élaboration collatérale (évaluation dans un ordre quelconque) des expressions $e_1,...,e_n$ qui sont les paramètres réels de l'appel. La règle de l'évaluation collatérale dit que $(e_1,...,e_n)$ a la valeur x si et seulement si elle a cette valeur quelque soit l'ordre d'évaluation des sous expressions atomiques la constituant.

Le prémisse 2 est la spécification de la procédure p. Si l'assertion d'entrée P est vraie alors soit l'appel à $p(x)$ satisfait l'assertion de sortie du cas normal $(Q \wedge \sigma_0 = v)$, soit il signale l'exception $ex(z)$ conditionnée par $E(z)$. Cela nécessite que tous les traitants de ex satisfassent la condition de reprise $(R(z) \wedge \sigma_1 = u)$.

Le prémisse 3 est la spécification du traitant h admettant le vecteur z pour paramètre formel. L'assertion d'entrée pour h comprend la condition $E(z)$ de l'exception qui lui est associée. Le traitant peut entrainer la reprise ou le remplacement du signalant. Dans le premier cas (l'expression se termine par END ou NO), il doit être montré que le traitant établit la condition $R(z) \wedge \sigma_1 = u$. Dans le second cas (l'expression est suivie de REPLACE), la condition de remplacement $(S \wedge \sigma_2 = w)$ doit être vérifiée.

La conclusion de la règle dit que si tous les prémisses sont vérifiés avant l'appel à p, alors après l'appel, l'assertion de sortie du cas normal ou l'assertion de sortie de remplacement du traitant est vérifiée. Nous étudions maintenant la règle dans le cas où le signalant est une construction fermée. Il s'agit d'un cas particulier de la règle précédente où il n'y a pas d'expressions paramètres.

R3 - Règle de l'appel d'une commande :

$$\frac{1.s\ PC\ wrt\ (N/\{P, Q \wedge \sigma_0 = v, ex(z) < E(z), R(z) \wedge \sigma_1 = u >\}) \quad 2.N/E(z)\{h(z)\}(R(z) \wedge \sigma_1 = u) \vee (REPLACE : S \wedge \sigma_2 = w)}{N/P\{s\ ON\ ex = \ h\ NO\}(Q \vee S) \wedge \sigma = (Q|v) \oplus (S|w)}$$

Nous illustrons cette proposition par un exemple [Yemi87].

Preuve de la correction de l'exemple 2.7 : Nous redonnons le corps de la procédure *Convert* :

```
PROC Convert = (REF [] INT code) STRING
SIGNALS (EXC (INT) (CHAR, STRING) code_incorrect) :
BEGIN
   FOR i FROM LWB  code TO UPB  code DO
      s:= s + Repr(code[i])
   OD
   ON
      car_inexistant = (CHAR, CHAR) : code_incorrect(i) REPLACE
   NO;
   s
END
```

Nous faisons la preuve de la procédure *Convert* dans le cas où le traitant rend un caractère de remplacement. Nous donnons tout d'abord la spécification de la procédure *Repr* :

Spécification de Repr :

$$\{P_{Repr(n)},\ Q_{Repr(n)},\ car\_inexistant < E_{car\_inexistant}\ ,\ R_{car\_inexistant} >\}$$

où

$$\begin{array}{lcl} P_{Repr(n)} & \equiv & Vrai \\ Q_{Repr(n)} & \equiv & min\_ent \leq n \leq max\_ent \wedge \sigma = repr\_car(n) \\ E_{car\_inexistant} & \equiv & (n > max\_ent) \vee (n < min\_ent) \\ R_{car\_inexistant} & \equiv & Q_{Repr(n)} \end{array}$$

Nous supposons que la procédure *Repr* a été prouvée correcte avec la spécification proposée. *repr_car* a la même signification que dans le paragraphe précédent.

La spécification de *Convert* utilise la relation d'affectation d'ALGOL 68 qui associe les identificateurs à leur valeur. Puisque *code* est un identificateur, il est lié à l'adresse d'un tableau d'entiers d_{code}. Le contenu de cette adresse est obtenu

par la translation τ et est noté v_{code}. *AFFECTER* et τ sont liés à la notion d'environnement et de mémoire de la sémantique dénotationnelle.

Spécification de Convert :

$$\{P_{Convert(code)},\ Q_{Convert(code)}, code\_incorrect(i) \\ < E_{code\_incorrect(i)}, R_{code\_incorrect(i)} >\}$$

où

$$\begin{array}{lcl} P_{Convert(code)} & \equiv & AFFECTER(code, d_{code}) \wedge \tau\ (d_{code}) = v_{code} \\ Q_{Convert(code)} & \equiv & P_{Convert(code)} \wedge \tau(d_s) = \\ & & +_{j=inf(code)}^{sup(code)}((min\_ent \leq v_{code}(j) \leq max\_ent | repr\_car(v_{code}(j))) \\ & & \oplus(((v_{code}(j) < min\_ent) \vee (v_{code}(j) > max\_ent))|\ car\_rempl)) \\ & & \wedge \sigma = d_s \\ E_{code\_incorrect(i)} & \equiv & P_{Convert(code)} \wedge\ ((v_{code}(i) < min\_ent) \vee (v_{code}(i) > max\_ent)) \\ R_{code\_incorrect(i)} & \equiv & E_{code\_incorrect(i)} \wedge\ \sigma = car\_rempl \end{array}$$

Nous commentons cette spécification. v_{code} est supposée être la valeur du contenu de l'adresse passée en argument de *Convert*. Quand *Convert* termine, l'argument est inchangé et la valeur retournée est l'adresse du résultat de la concaténation successive des caractères de représentation ou de remplacement selon que l'élément du tableau est représentable ou non. L'exception *badcode(i)* est signalée quand le i^{eme} du tableau *code* n'est pas représentable. Le traitant de cette exception doit fournir un caractère de remplacement de manière à ce que *Convert(code)* satisfasse $Q_{Convert(code)}$. *car_rempl* désigne le caractère de remplacement.

Preuve :

Pour simplifier la notation, nous introduisons la fonction *chaine(inf, sup)* définie comme suit :

$$\begin{array}{l} chaine(inf,\ sup) \equiv +_{j=inf}^{sup} \\ \qquad ((min\_ent \leq v_{code}(j) \leq max\_ent|\ repr\_car(v_{code}(j))) \\ \qquad \oplus(((v_{code}(j) < min\_ent) \vee (v_{code}(j) > max\_ent))|\ car\_rempl)) \end{array}$$

La principale étape de la preuve de la correction de *Convert* avec sa spécification est :

$$P_{Convert(code)} \wedge AFFECTER(s, d_s) \wedge \tau(d_s) = ""$$

```
{ FOR i FROM LWB code TO UPB code DO s:= s + Repr(code[i]) OD
ON car_inexistant = (CHAR, CHAR) : code_incorrect(i) REPLACE NO; }
```

$$P_{Convert(code)} \wedge AFFECTER(s, d_s) \wedge \tau(d_s) = chaine(LWBcode, UPBcode) \wedge \sigma = vide$$

La preuve de la correction de la procédure *Convert* nécessite la règle de preuve de l'itération :

$$\frac{1.N/B \Rightarrow I([]) \quad 2.N/i' \in [\text{inf}, \text{sup}] \wedge I([\text{inf}, i'[)\{corps\}I([\text{inf}, i'])}{N/B\{FOR\ i\ FROM\ inf\ TO\ sup\ DO\ corps\ OD\}I([\text{inf}, \text{sup}]) \wedge \sigma = vide}$$

où I désigne l'invariant de boucle.

Posons :

$$\begin{array}{lcl} S & \equiv & P_{convert(code)} \wedge AFFECTER(s, d_s) \\ B & \equiv & S \wedge \tau(s) = \text{""} \\ I([inf, sup]) & \equiv & S \wedge \tau(d_s) = chaine(inf, sup) \end{array}$$

La preuve du premisse 1 de la règle de preuve de l'itération est immédiate. Pour montrer le prémisse 2, la règle d'appel d'un signalant (R2) est nécessaire pour obtenir l'effet de *Repr*. Nous devons montrer :

$$I([LWB\ code, i'[)$$

```
{ Repr(code[i])
ON car_inexistant=(CHAR,CHAR):code_incorrect(i) REPLACE NO }
```

$$I([LWB\ code, i'[) \wedge$$
$$\sigma = +_{j=LWBcode}^{i'}((min\_ent \leq v_{code}(j) \leq max\_ent | repr\_car(v_{code(j)}))) \oplus$$
$$(min\_ent > v_{code}(j) \vee v_{code}(j) > max\_ent | car\_rempl))$$

Nous examinons les prémisses de R2. La règle de l'appel est utilisée avec :

$$T \equiv I([LWBcode, i'[).$$

Puisqu'il n'y aucun effet de bord dans l'évaluation de $code[i]$, le résultat du prémisse 1 est :

$$1.N/I([LWB\ code, i'[)\{\texttt{code[i]}\}I([LWB\ code, i'[) \wedge \sigma = v_{code}(i)$$

Pour ce qui est du prémisse 2, la procédure *Repr* est supposée cohérente avec la spécification donnée.

Le prémisse 3 nécessite l'examen du traitant de l'exception *car_inexistant*. Ce traitant propage l'exception *car_inexistant* comme l'exception *code_incorrect* de *Convert*. En prouvant la correction partielle d'un signalant, la définition de la correction partielle permet de supposer que tout traitant, pour une exception signalée dans le corps du signalant, est partiellement correct étant données les conditions de reprise et de signal de l'exception. Par conséquent, si l'on prouve que $E_{code\_incorrect(i)}$ est vraie juste avant le signal de *code_incorrect(i)* alors on peut supposer que la condition de reprise $R_{code\_incorrect(i)} \equiv E_{code\_incorrect(i)} \wedge \sigma = car\_rempl$ est vraie juste après le signal. On obtient en appliquant R1 :

$$E_{car\_inexistant}$$

```
{ (CHAR, CHAR) : code_incorrect(i) REPLACE }
```

$$REPLACE : P_{Convert(code)} \wedge (v_{code}(i) > max\_ent \vee v_{code}(i) < min\_ent) \wedge \sigma = car\_rempl$$

En appliquant la règle R2, on peut conclure que :

$$I([LWB\ code, i']) \wedge \sigma = ((min\_ent \leq v_{code}(i) \leq max\_ent | repr\_car(v_{code}(i)) \oplus ((v_{code}(i) > max\_ent) \vee (v_{code}(i) < min\_ent) | car\_rempl)$$

est vérifié après l'appel à *Repr*.

Par conséquent :

$$I([LWBcode, i'[)$$

```
{ s:=s+Repr(code[i]);
ON car_inexistant = (CHAR,CHAR): code_incorrect(i) REPLACE NO }
```

$$I([LWBcode, i']) \wedge \sigma = d_s$$

En appliquant la régle de l'itération, on en déduit l'assertion de sortie :

$$I([LWBcode, UPBcode]) \equiv P_{Convert(code)} \wedge AFFECTER(s, d_s) \wedge \tau(d_s) = chaine(LWBcode, UPBcode)$$

La dernière expression du corps de *Convert* est s, dont la valeur est retournée comme le résultat de la procédure.

$$I([LWBcode, UPBcode])\{$$
$$\mathbf{s}$$
$$\}P_{Convert(code)} \wedge$$
$$AFFECTER(s, d_s) \wedge$$
$$\tau(d_s) = chaine(LWB\ code, UPB\ code) \wedge$$
$$\sigma = d_s$$

Cette preuve s'obtient immédiatement à partir de la règle d'élaboration d'un identificateur. Puisque la postcondition de cette preuve implique immédiatement $Q_{Convert(code)}$, on peut conclure que la procédure *Convert* est cohérente avec sa spécification. $\diamond$

Le MTE que nous présentons dans le paragraphe suivant est plus simple. Mais, il introduit une axiomatique fondée sur l'utilisation de la formulation de la précondition la plus faible.

4.2 Axiomes fondé sur l'utilisation de la formulation "wp"

A notre connaissance, le seul système d'axiomes d'un MTE fondé sur la formulation *wp* est celui proposé dans [Cris83] [Cris84]. L'évaluation de la fonction sémantique *wp(S, R)* rend la plus faible précondition telle que l'exécution de S termine certainement et telle que le résultat vérifie R.

4.2.1 Sémantique axiomatique d' un MTE de type terminaison

Le concept de transformateur de prédicat a été introduit par Dijkstra pour spécifier la sémantique des commandes à une Entrée et une Sortie. Ceci permet de savoir quand la sortie standard associée est atteinte et quelle transformation d'états s'est produite. Dans le cas d'un programme contenant des exceptions, il apparait qu'il faut autant de transformateurs que de points de sortie [Cris84]. On trouve, par conséquent, le transformateur standard noté *wp(C, fin , Q)* qui correspond à celui de Dijkstra, et un transformateur par point de sortie exceptionnel e_i noté *wp(C, e_i , Q)*. Pour savoir si une commande termine au point de sortie x, exceptionnel ou non, il suffit de dériver *wp(c, x, Vrai)*. De plus, en donnant des règles de dérivation de *wp(C, x, Q)*, on sait quelles transitions se sont produites entre l'entrée et la sortie de C. Un programme P avec k points de sortie exceptionnels est *robuste* s'il termine à un point de sortie déclaré pour tout état d'entrée possible. Plus formellement, la propriété de robustesse est la suivante.

P1 - Robustesse :

$$wp(P, fin, Vrai) \vee (\vee_{i=1}^{k}(wp(P, e_i, Vrai))) = Vrai$$

Afin de spécifier le comportement attendu d'un programme P qui signale k exceptions, il faut $(k+1)$ pré- et post-conditions [Cris82b]. On obtient un ensemble de couples de la forme (R_i, S_i). Le couple (R_0, S_0) dénote le comportement standard attendu de P, et les couples (R_j, S_j), $j > 0$, spécifient le comportement exceptionnel attendu de P. Un programme P, est **correct** étant donné une telle spécification, si la propriété suivante est vérifiée.

P2 - Correction :

$$(R_0 \Rightarrow wp(P, fin, S_0)) \ \&(\&_{i=1}^{k} R_i \Rightarrow wp(P, e_i, S_i))$$

Par conséquent, la preuve de correction d'un programme pouvant signaler k exceptions consiste en $(k+1)$ sous-preuves indépendantes. Il faut maintenant définir la fonction sémantique *wp* pour l'ensemble des constructions du langage. Le domaine de cette fonction est :

$$wp : COMMANDE \times POINT\_DE\_SORTIE \times PREDICAT \rightarrow PREDICAT$$

wp(C, x, Q) est la plus faible précondition qui garantit que C terminera en x dans un état satisfaisant Q. La sémantique d'une commande C se définit naturellement par induction syntaxique sur la structure de la commande. Nous introduisons tout d'abord la syntaxe du langage [Cris83].

Un programme écrit dans ce langage consiste en un nom, un ensemble de déclarations et une commande C.

PROG ::= *prog* N [EL] VDECL; PDECL; C.

Les déclarations donnent l'ensemble des points de sortie e exceptionnels dans lesquels N peut terminer. Elles donnent aussi les variables v et les procédures Pr connues dans N.

EL ::= e | e; EL.
VDECL ::= | *var* v : int; VDECL.
PDECL ::= | *proc* Pr(v vp, vr vrp, r rp) [EL] VDECL; C; PDECL.

Nous définissons C. Dans un souci de simplicité, seules les variables de type entier sont considérées.

C ::= | signal e | S | CP | C; C.
S ::= v:= E | if B then C else C fi | while B do C od | begin C end | Pr(va, vra, ra).
CP ::= [B:C] | S[LT].
LT ::= EL : C | EL : C, LT.
E ::= m | v | (E) | E+E | E−E | E × E | E *div* E.
B ::= Expressions booléennes (Non détaillé)

Les domaines syntaxiques qui nous intéressent plus particulièrement sont CP (pour Construction Protégée) et LT (pour Liste de Traitants). Il y a deux types de constructions protégées. La première permet d'appeler un traitant C si l'expression booléenne B est vraie. La seconde permet d'associer un ou plusieurs traitants à une commande simple S. Enfin, LT permet de définir les traitants pour les points de sortie exceptionnels.

Nous donnons la définition sémantique des commandes se rapportant au traitement d'exceptions [Cris83].

S1 - Commande vide :

$$\begin{aligned} wp(,fin,Q) &\equiv Q \\ wp(,e,Q) &\equiv Faux \end{aligned}$$

La commande vide termine toujours normalement sans changer l'état du programme.

S2 - Commande de signal :

$$\begin{aligned} wp(signal\ e,fin,Q) &\equiv Faux \\ wp(signal\ e,e,Q) &\equiv Q \end{aligned}$$

Quand *signal* e est exécutée, le point de sortie devient e.

S3 - Composition séquentielle :

$$\begin{aligned} wp(C_1;C_2,fin,Q) &\equiv wp(C_1,fin,wp(C_2,fin,Q)) \\ wp(C_1;C_2,e,Q) &\equiv wp(C_1,e,Q) \vee wp(C_1,fin,wp(C_2,e,Q)) \end{aligned}$$

$C_1;C_2$ termine normalement si C_1 et C_2 terminent normalement. $C_1;C_2$ termine exceptionnellement au point e si C_1 ou C_2 signale e.

S4 - Appel de procédure :

$$wp(Pr(fi,fio,fo),x,Q) \equiv wp(init\_proc,fin,wp(Corps,x,wp(fin\_proc,fin,Q)))$$

où $\begin{aligned} &init\_proc \equiv vp := va;\ vrp := vra \\ &\text{et } fin\_proc \equiv vra := vrp;\ ra := rp. \end{aligned}$

Le corps de la procédure Pr est exécuté après le prélude *init_proc*. Une fois la procédure terminée en un point de sortie x, le postlude *fin_proc* est exécuté.

S5 - Construction protégée :

$$
\begin{array}{lcl}
wp([B : C];\ fin;\ Q) & \equiv & \neg B\ \&\ Q \vee B\ \&\ wp(C, fin, Q) \\
wp([B : C];\ e;\ Q) & \equiv & B\ \&\ wp(C,\ e,\ Q)
\end{array}
$$

Une construction protégée $[B : C]$ termine normalement si B est évaluée à faux (le traitant n'est pas exécuté) ou si B est évaluée à vrai et si C termine normalement. La construction peut signaler e si B est évaluée à vrai et si C signale e.

$$
\begin{array}{lcl}
wp(S[e1 : C1, ..., en : Cn], fin, Q) & \equiv & wp(S, fin, Q) \vee (\vee_{i=1}^{n}(wp(S, ei, wp(Ci, fin, Q))) \\
wp(S[e1 : C1, ..., en : Cn], e, Q) & \equiv & (e \in \{e1, ..., en\}) \rightarrow \\
 & & (\vee_{i=1}^{k}(wp(S, e_i, wp(C_i, e, Q)))); \\
 & & wp(S, e, Q) \vee (\vee_{i=1}^{k}(wp(S, ei, wp(Ci, e, Q))))
\end{array}
$$

Une construction protégée de la forme $S[e1 : C1, ..., en : Cn]$ termine normalement si la commande simple S termine normalement, ou si S signale une exception e_i dont le traitant C_i termine normalement. Cette construction signale e si S signale e, et si e n'est pas traitée localement, ou si S signale une exception dont le traitant C_i signale e.

S6 - Propagation explicite :

$$wp(P[e\ :\ C;\ signal\ f], fin, Q) \quad \equiv \quad wp(P, fin, Q)$$

Nous reprenons l'exemple 2.4 avec cette nouvelle syntaxe.

Exemple 4.3 : Nous nous intéressons à l'appel de la procédure *Convert* qui est de la forme suivante :

$$P \equiv Convert(code, s)[code\_incorrect : s := ""]$$

Nous introduisons les types *char*, *string* et *array*. La procédure *Convert* est définie comme suit :

```
proc Convert(v code:array[1..n] of int; vr s:string) [code_incorrect]
var i:int;
var c:char;
begin
  i:=0;
  while i<n do
    i:=i+1; Repr(code[i], c); s:=s+c
  od [car_inexistant:signal code_incorrect]
end
```

◇

Nous proposons un exemple de preuve de correction.

Preuve de correction de l'exemple 4.3 : Nous voulons prouver que la procédure *Convert* satisfait les spécifications F1 et F2 suivantes :

Spécification de Convert :

F1 - Spécification du cas normal de *Convert* :

$$\begin{gathered} \forall i : 1 \leq i \leq n, min\_ent \leq code[i] \leq max\_ent \\ \{\texttt{Convert(code, s)}\ \} \\ fin : \forall i : 1 \leq i \leq n, s[i] = repr\_car(code[i]) \end{gathered}$$

F2 - Spécification du cas exceptionnel de *Convert* :

$$\begin{gathered} \exists i : 1 \leq i \leq n, ((code[i] > max\_ent) \vee (code[i] < min\_ent)) \\ \{\ \texttt{Convert(code, s)}\ \} \\ code\_incorrect : Vrai \end{gathered}$$

Nous faisons certaines hypothèses de simplification. Tout d'abord, la procédure *Repr* est supposée correcte avec les spécifications HF1 et HF2 suivantes.

Spécification de Repr :

HF1 - Spécification du cas normal de *Repr* :

$$\begin{gathered} (min\_ent \leq ent \leq max\_ent) \\ \{\ \texttt{Repr(ent, car)}\ \} \\ fin : car = repr\_car(ent) \end{gathered}$$

HF2 - Spécification du cas exceptionnel de *Repr* :

$$\begin{gathered} ((ent > max\_ent) \vee (ent < min\_ent)) \\ \{\ \texttt{Repr(ent, car)}\ \} \\ car\_inexistant : Vrai \end{gathered}$$

Hypothèses :

Nous supposons aussi les équivalences suivantes :

H1 - Cas normal :

$$\begin{array}{lcl} wp(i+j, fin, Q) & \equiv & Q \\ wp(init\_proc, fin, Q) & \equiv & Q \\ wp(fin\_proc, fin, Q) & \equiv & Q \\ wp(a := b, fin, Q) & \equiv & Q[b/a] \end{array}$$

H1' - Cas exceptionnel :

$$\begin{array}{lcl} wp(i+j,e,Q) & \equiv & Faux \\ wp(init\_proc,e,Q) & \equiv & Faux \\ wp(fin\_proc,e,Q) & \equiv & Faux \\ wp(a:=b,e,Q) & \equiv & Faux \end{array}$$

Nous devons aussi donner la définition sémantique de l'itération. D'une manière générale, il est difficile de trouver une condition nécessaire et suffisante à la terminaison d'une itération. Aussi, nous utilisons deux théorèmes [Cris84] qui permettent de montrer qu'une condition I est suffisante à la terminaison d'une itération avec une postcondition Q. I et V désignent respectivement l'invariant et le variant de la boucle. V est à valeurs entières.

Théorème 1 - Cas normal :

$$\frac{\begin{array}{l} 1.I\&\neg B \Rightarrow Q, \\ 2.I\&(V \leq 0) \Rightarrow \neg B, \\ 3.I\&B\&(V \leq t) \Rightarrow wp(C,fin,I\&V < t) \end{array}}{I \Rightarrow wp(while\ B\ do\ C\ od,fin,Q)}$$

Théorème 2 - Cas exceptionnel :

$$\frac{\begin{array}{l} 1.I \Rightarrow B, \\ 2.I\&(V \leq 0) \Rightarrow wp(C,e,Q), \\ 3.I\&(V > 0)\&(V \leq t) \Rightarrow wp(C,fin,I\&V < t) \end{array}}{I \Rightarrow wp(while\ B\ do\ C\ od,e,Q)}$$

Etant donnée la propriété de correction (P2), nous devons prouver F3 et F4.

Correction de Convert :

F3 - Correction dans le cas normal :

$$\begin{array}{c} 0 \leq n, \forall i : 1 \leq i \leq n, min\_ent \leq code[i] \leq max\_ent \Rightarrow \\ wp(Convert(code,s),fin,\forall i : 1 \leq i \leq n, s[i] = repr\_car(code[i])) \end{array}$$

F4 - Correction dans le cas exceptionnel :

$$\begin{array}{c} 0 \leq n, \exists i : 1 \leq i \leq n, ((code[i] > max\_ent) \vee (code[i] < min\_ent)) \Rightarrow \\ wp(Convert(code,s),code\_incorrect,Vrai) \end{array}$$

Une fois, F3 et F4 prouvées, nous pouvons aussi conclure que *Convert* est robuste (P1). En effet, la disjonction des prémisses de F3 et F4 est vraie ou encore:

$$(0 \leq n, \forall i : 1 \leq i \leq n, min\_ent \leq code[i] \leq max\_ent)$$
$$\vee$$
$$(0 \leq n, \exists i : 1 \leq i \leq n, ((code[i] > max\_ent) \vee (code[i] < min\_ent)))$$
$$\equiv$$
$$Vrai$$

On a donc $wp(Convert(code, s), fin, \forall i : 1 \leq i \leq n, s[i] = repr\_car(code[i])) \vee wp(Convert(code, s), code\_incorrect, Vrai)$.

Nous prouvons tout d'abord F3 et nous simplifions la notation en écrivant :

$$\text{F3} \equiv R0 \Rightarrow wp(Convert(code, s), fin, S0).$$

Preuve de F3 :

D'après H3 et S4 (*iter_convert* désigne l'itération dans le corps de la procédure *Convert*) :

$wp(Convert(code, s), fin, S0) \equiv$
$wp(i := 0; iter\_convert[nochar : signal\ code\_incorrect, fin, S0)$

D'après S3 et S6 :

$wp(i := 0; iter\_convert[nochar : signal\ code\_incorrect], fin, S0) \equiv$
$wp(i := 0, fin, wp(iter\_convert, fin, S0))$ (*)

Si nous considérons le théorème 1, nous devons évaluer *wp(i:=0, fin, I)* car en prouvant les prémisses de ce théorème, on obtient $I \Rightarrow wp(iter\_convert, fin, S0)$. Nous donnons I et V :

$$\begin{aligned} I &\equiv 0 \leq n \ \& \ i \leq n \ \& \\ &\quad \forall j : 1 \leq j \leq i,\ s[j] = repr\_car(code[j]) \\ V &\equiv n - i \end{aligned}$$

De plus, nous prenons :

$$\begin{aligned} B &\equiv i < n \\ Q &\equiv S0 \end{aligned}$$

Nous examinons les trois prémisses du théorème 1 :

1. $I \ \& \ \neg B \equiv$
$0 \leq i \ \& \ i \leq n \ \& \ \forall j : 1 \leq j \leq i, s[j] = repr\_car(code[j]) \ \ \& \ \neg(i < n) \Rightarrow$
$0 \leq i \ \& \ i \leq n \ \& \ n \leq i \ \& \ \forall j : 1 \leq j \leq i,\ s[j] = repr\_car(code[j]) \Rightarrow$
$\forall j : 1 \leq j \leq n, s[j] = repr\_car(code[j]) \ \& \ \neg(i < n) \Rightarrow$

$S0 \equiv \mathbf{Q}$

2. $I \,\&\, V \leq 0 \equiv$
$0 \leq i \,\&\, i \leq n \,\&\, \forall j : 1 \leq j \leq i,\, s[j] = repr\_car(code[j]) \,\&\, (n-i) \leq 0 \Rightarrow$
$i \leq n \,\&\, n \leq i \Rightarrow$
$i = n \Rightarrow$
$\neg(i < n) \Rightarrow \neg\, \mathbf{B}$

3. $I \,\&\, B \,\&\, V \leq t \equiv$
$0 \leq i \,\&\, i \leq n \,\&\, i < n \,\&\, \forall j : 1 \leq j \leq i,\, s[j] = repr\_car(code[j]) \,\&\, (n-i) \leq t \Rightarrow$
$0 \leq i \,\&\, i < n \,\&\, \forall j : 1 \leq j \leq i, s[j] = repr\_car(code[j]) \;\&\, (n-i) \leq t$

Nous évaluons $wp(corps\_iter\_convert, fin, I\&V < t)$:

D'après S3 :

$wp(corps\_iter\_convert, fin, I\&V < t) \equiv$
$wp(i := i+1, fin, wp(Repr(code[i], c); s := s + c, fin, I\&V < t))$

D'après H1 et S3 :

$wp(i := i+1, fin, wp(Repr(code[i], c); s := s + c, fin, I\&V < t)) \equiv$
$wp(Repr(code[i], c), fin, wp(s := s + c, fin, I\&V < t))[i/i+1]$

D'après HF1, H1, et
$wp(Repr(ent, car), fin, car = repr\_car(ent)) \equiv min\_ent \leq ent \leq max\_ent$:

$wp(Repr(code[i], c), fin, wp(s := s + c, fin, I\&V < t))[i/i+1] \equiv$
$min\_ent \leq code[i] \leq max\_ent \& (I\&V < t)[i+1/i][s+c/s] \Rightarrow$
$0 \leq (i+1) \,\&\, (i+1) \leq n \&$
$\forall j : 1 \leq j \leq i+1,\, s[j] = repr\_car(code[j]) \&$
$(n - (i+1)) < t$

Nous pouvons conclure :

$I\&B\&V \leq t \Rightarrow wp(corps\_iter\_convert,\, fin, I\&V < t)$

Par conséquent : $I \Rightarrow wp(iter\_convert, fin, S0)$

Nous revenons à l'étape (*) et nous évaluons : $wp(i := 0, fin, I)$

D'après H1 :

$wp(i := 0, fin, I) \equiv I[0/i]$

$R0 \Rightarrow I$ permet d'assurer : F3 est vérifiée.

Nous prouvons F4.

Preuve de F4 :

Après S4 et H1, il s'agit d'évaluer $wp(corps\_convert, code\_incorrect, Vrai)$.

D'après S5 :

$wp(corps\_convert, code\_incorrect, Vrai) \equiv$
$wp(i := 0; iter\_convert, code\_incorrect, Vrai) \vee$
$wp(i := 0; iter\_convert, car\_inexistant, wp(signal\ code\_incorrect, code\_incorrect, Vrai)$

Nous ne prouvons pas $wp(i := 0; iter\_convert, code\_incorrect, Vrai) \equiv Faux$.
D'après S2, S3, H1, H1', et
$wp(i := 0; iter\_convert, code\_incorrect, Vrai) \equiv Faux$,
on obtient :

$wp(i := 0; iter\_convert, code\_incorrect, Vrai) \vee$
$wp(i := 0; iter\_convert, car\_inexistant, wp(signal\ code\_incorrect, code\_incorrect, Vrai) \equiv$

$wp(i := 0, car\_inexistant, Vrai) \vee wp(i := 0, fin, wp(iter\_convert, car\_inexistant, Vrai)) \equiv$

$wp(i := 0, fin, wp(iter\_convert, car\_inexistant, Vrai)) \equiv$
$wp(iter\_convert, car\_inexistant, Vrai)[0/i]$

D'après H1 et le théorème 2. Soit I l'invariant de boucle, nous devons montrer :

$wp(i := 0, fin, I) \equiv I[0/i]$

B et Q sont inchangés. Nous donnons I & V.

$$\begin{array}{lcl} I & \equiv & 0 \leq i \ \& \ i \leq k \ \& \ k < n \ \& \\ & & \forall j : 1 \leq j < i,\ s[j] = repr\_code(code[j]) \ \& \\ & & code[k] > max\_ent \vee code[k] < min\_ent \\ V & \equiv & k - i \end{array}$$

k est l'indice du premier élément non représentable dans le tableau *code*. $I[i/0]$ est vrai. Nous admettons les prémisses 1 et 3, et nous prouvons le prémisse 2 du théorème 2 :

1. $I \Rightarrow B$

3. $I\&(V > 0)\&(V \leq t) \Rightarrow wp(corps\_iter\_convert, fin, V < t)$

2. $I\&V \leq 0 \equiv$
$0 \leq i \ \& \ i \leq k \ \& \ k < n \ \& \ \forall j : 1 \leq j \leq i, s[j] = repr\_code(code[j]) \ \& \ (k - i) \leq 0$

D'après S3 :

$wp(corps\_iter\_convert, car\_inexistant, Vrai) \equiv$
$wp(i := i + 1, car\_inexistant, Vrai) \vee$
$wp(i := i + 1, fin, wp(Repr(code[i], c); s := s + c, car\_inexistant, Vrai))$

D'après H1 et S3 :

$wp(i := i + 1, car\_inexistant, Vrai) \vee$
$wp(i := i + 1, fin, wp(Repr(code[i], c); s := s + c, car\_inexistant, Vrai)) \equiv$
$wp(Repr(code[i], c), car\_inexistant, Vrai)[i/i + 1] \vee$
$wp(Repr(code[i], c), fin, wp(s := s + c, car\_inexistant, Vrai))[i/i + 1]$

D'après $V \leq 0$:

$wp(Repr(code[i], c), car\_inexistant, Vrai)[i/i + 1] \vee$
$wp(Repr(code[i], c), fin, wp(s := s + c, car\_inexistant, Vrai))[i/i + 1] \equiv$
$wp(Repr(code[i], c), car\_inexistant, Vrai)$

D'après SH1 et I et $V \leq 0$ et $(I \wedge (V \leq 0) \Rightarrow i = k)$:

$wp(Repr(code[i], c), car\_inexistant, Vrai) \equiv$
$code[i] > max\_ent \vee code[i] < min\_ent$

Nous pouvons conclure :

$I\&V \leq 0 \Rightarrow wp(corps\_iter\_convert, car\_inexistant, Vrai)$.

Par conséquent :

$I \Rightarrow wp(iter\_convert, car\_inexistant, Vrai)$

$(\exists i : 1 \leq i \leq n, code[i] > max\_ent \vee code[i] < min\_ent \Rightarrow I)$ permet d'assurer : F4 est vérifiée.

La procédure *Convert* est correcte et robuste $\diamond$

Nous terminons ce paragraphe par la présentation d'un système de déduction fondé sur la logique de Hoare [Cris84]. Il permet de prouver les propriétés de correction totale et de robustesse de programmes contenant des exceptions. De plus, il nous semble plus simplement manipulable que les systèmes introduits dans le paragraphe 4.1. Ce système comprend des axiomes, un pour chaque exception prédéfinie du langage, et des règles d'inférence pour chaque construction syntaxique du langage. Ces règles sont déduites des propriétés suivantes.

$$Si\ (P \Rightarrow Q)\ Alors\ (wp(C,x,P) \Rightarrow wp(C,x,Q))$$
$$wp(C,x,P\&Q) \equiv wp(C,x,P)\&wp(C,x,Q)$$
$$wp(C,x,P \vee Q) \equiv wp(C,x,Q) \vee wp(C,x,Q)$$

Nous n'introduisons que quelques unes des règles pour lesquelles nous avons donné la définition de la fonction *wp*. Une proposition de la forme $P\{C\}x{:}Q$ signifie que si C est exécutée dans un état dans lequel P est vraie alors C termine au point x dans un état dans lequel Q est vraie. Par convention, si $x = fin$, on écrit $P\{C\}Q$ au lieu de $P\{C\}fin{:}Q$.

A1 - Commande vide :

$$P\{\}P$$

A2 - Commande de signal :

$$P\{signal\ e\}e : P$$

R1 - Appel de procédure :

$$\frac{P\{init\_proc\}Q, Q\{corps\}x : R, R\{fin\_proc\}S}{P\{Pr(va,vra,ra)\}x : S}$$

R2 - Commande signalante S :

$$\frac{e \in \{e_1,...,e_n\}, P\{S\}e_i : R, R\{C_i\}x : Q}{P\{S[e_1 : C_1,...,e_n : C_n]\}x : Q}$$

$$\frac{x \notin \{e_1,...,e_n\}, P\{S\}x : Q}{P\{S[e_1 : C_1,...,e_n : C_n]\}x : Q}$$

Enfin, nous pouvons citer une étude reposant sur une extension de la formulation *wp* [Best81]. Elle permet, étant donnés un programme et sa spécification, de déterminer le domaine d'entrée standard pour lequel le programme rend le résultat attendu et le domaine d'entrée exceptionnel pour lequel le programme termine dans un état exceptionnel. La méthode proposée permet aussi de savoir où les tests de détection d'exceptions doivent être insérés dans le programme.

Après avoir présenté le traitement des exceptions dans les programmes, nous nous intéressons au traitements des exceptions au niveau des données. L'objet du chapitre suivant est le traitement des exceptions dans les types abstraits algébriques.

5 Traitement d'exceptions dans les types abstraits algébriques

Les types abstraits algébriques permettent de caractériser les données. Le traitement des erreurs dans les types abstraits algébriques permet notamment de spécifier des structures bornées telles que les tableaux ou les piles bornées. Avant de présenter deux formalismes admettant le traitement des exceptions, nous faisons une brève introduction aux types abstraits algébriques. Nous concluons ce chapitre par une étude de deux langages de spécification algébrique prenant en compte les exceptions.

5.1 Types abstraits algébriques

Nous présentons brièvement les types abstraits algébriques [Gogu78] puis nous nous interrogeons sur l'utilité de définir un nouveau formalisme pour traiter les exceptions.

Les types de données sont des algèbres où les données représentées sont les éléments du support et où les fonctions accessibles à l'utilisateur sont les opérations d'une algèbre. D'une manière générale, les algèbres sont hétérogènes car il y a plusieurs sortes de données impliquées (une sorte peut être vue comme un type de données). On définit un type de données au moyen d'une signature. Cette signature fournit les sortes, les noms d'opérations ainsi que les sortes des arguments et des résultats de chaque opération. Mathématiquement, une algèbre est un couple <S, Σ > où S est l'ensemble des sortes, et Σ une signature sur S. Σ est de la forme $< \Sigma_{u,v} >_{u,v \in S}$ où un $\Sigma_{u,v}$ est l'ensemble des symboles des opérations de u dans v. Une Σ-algèbre est une famille S-indexée $A=<A_s>_{s \in S}$ d'ensembles, et une fonction $A_\sigma : A_u \rightarrow A_v$ est définie pour chaque symbole opération $\sigma \in \Sigma_{u,v}$. A_s est appelé support de la sorte S. Nous donnons un exemple simple.

Exemple 5.1 : Il s'agit d'un extrait de la signature et des équations de pile(entier). p désigne la pile, e les entiers, b les booléens. Lorsque l'arité d'une opération est nulle (il s'agit d'une constante), nous utilisons la notation λ.

S = {p, e, b};

$\Sigma_{\lambda,p}$ = {VIDE},
$\Sigma_{\lambda,e}$ = Z,
$\Sigma_{\lambda,b}$ = {V, F},
$\Sigma_{p,p}$ = {DEPILER},
$\Sigma_{ep,p}$ = {EMPILER},
$\Sigma_{p,e}$ = {SOMMET},
$\Sigma_{p,b}$ = {ESTVIDE?},
$\Sigma_{ee,e}$ = {+, x, -};

Equations

DEPILER(EMPILER(E, P)) = P
SOMMET(EMPILER(E, P)) = E
ESTVIDE?(EMPILER(E, P)) = F
ESTVIDE?(VIDE) = V
Plus les équations pour les opérations +, × et − sur Z.

Nous expliquons quelques unes des opérations proposées. VIDE est une constante qui désigne la pile vide. DEPILER est une opération de pile dans pile qui permet de retirer le premier élément de la pile. SOMMET est une opération de pile dans entier et rend le premier élément de la pile. Enfin, ESTVIDE? est une opération de pile dans booléen qui rend vrai si la pile est vide. ◇

Un type abstrait de données est indépendant de sa représentation puisque les détails de son implémentation sont masqués à l'utilisateur. La notion d'abstraction a le même sens que dans "algèbre abstraite". L'algèbre abstraite fournit un sens mathématique précis à la notion d'abstraction. Un point important de la spécification des types abstraits est la correction d'une implémentation : il faut être sûr que la spécification est correcte et que l'implémentation correspond à la spécification [Gogu75]. Il existe certains cas standards pour vérifier la correction de la spécification. Par exemple, on connait une définition des entiers et des listes.

Le traitement des exceptions dans les types abstraits algébriques est nécessaire si l'on veut, par exemple, pouvoir définir des structures bornées. Ceci étant, nous pouvons nous interroger sur la nécessité de définir un nouveau formalisme [Bern86a]. Une idée simple serait d'ajouter une constante d'erreur pour chaque sorte définie, avec le formalisme algébrique habituel. Par exemple, si l'on considère que les valeurs correctes d'une sorte sont celles comprises entre 0 et Entier_Max,

on posera :

$$PRED(0) = NONDEF \text{ et } SUCC(Entier\_Max) = NONDEF$$

Cette solution n'est pas satisfaisante puisqu'il n'y a pas complétude de la spécification. En effet, on ne sait pas comment traiter tous les termes au dessus de NONDEF comme SUCC(NONDEF) ou PRED(NONDEF). On peut alors penser à propager les erreurs en spécifiant :

$$PRED(NONDEF) = SUCC(NONDEF) = NONDEF$$

Mais, dans ce cas la spécification est inconsistante car ces deux axiomes nous amènent à déduire que tout entier naturel est égal à NONDEF. Pour s'en persuader, il suffit de considérer l'axiome PRED(SUCC(n))=n et l'évaluation de PRED(SUCC(Entier_max)). Pour éviter cette inconsistance, on peut imaginer l'ajout de l'axiome :

$$PRED(NONDEF)=Entier\_Max$$

Nous arrivons alors à une autre situation qui n'est pas non plus souhaitable à savoir :

$$PRED(PRED(0)) = PRED(NONDEF) = Entier\_max$$

Une dernière solution envisageable consiste à définir plusieurs valeurs erronées par sorte. Il se pose de nouveau un problème de complétude puisqu'il faut définir les opérations faisant intervenir plusieurs valeurs erronées. Dans le paragraphe suivant, nous présentons deux formalismes du traitement d'exceptions existants.

5.2 Deux formalismes du traitement d'exceptions

Encore une fois, il ne s'agit pas d'une énumération complète de tous les formalismes. Nous présentons plutôt une évolution du traitement des exceptions dans les types abstraits algébriques. Nous nous attardons sur deux formalismes. Les erreur-algèbres montrent qu'il est possible d'avoir des spécifications lisibles avec traitement d'exceptions. Les exceptions-algèbres ont l'avantage de permettre de spécifier la récupération des erreurs.

5.2.1 Les erreur-algèbres

Une erreur-algèbre [Gogu78] est composée de valeurs "Ok" (ou correctes) et de valeurs "erronées". Il existe par conséquent deux types d'opérations: les Ok-opérations et les Err-opérations. Une signature d'erreur est un triplet (S, Σ, Ξ) où Σ et Ξ sont des signatures sur S disjointes. Les opérateurs dans Σ sont les opérateurs Ok et ceux de Ξ sont les opérateurs erreur.

Une (S, Σ, Ξ)-algèbre erreur est une ($\Sigma \bigcup \Xi$)-algèbre sur S notée $A = <A_s>_{s \in S}$. Chaque support A_s est une union disjointe d'éléments Ok (K_s) et d'élèments Erreur (E_s) telle que:

1. Pout tout $\sigma \in \Sigma_{u,v}$, si un composant de a $\in A_u$ est dans un E_s alors A_σ(a) $\in E_v$

2. Pour tout $\xi \in \Xi_{u,v}$, et a $\in A_u$, A_ξ (a) $\in E_v$;

Cette définition permet d'assurer la propagation des erreurs puisque tout opérateur avec un argument "erreur" rend une erreur, et les opérateurs d'erreur engendrent toujours des éléments "erreur".

Une présentation avec erreurs (ou spécification) est un 5-uplet P=(S, Σ, Ξ, K, E) où (S, Σ, Ξ) est une signature d'erreur, K un ensemble de Σ-équations, et E un ensemble de ($\Sigma \bigcup \Xi$)- équations. Informellement, une algèbre ($\Sigma \bigcup \Xi$)-erreur A satisfait une présentation avec erreurs P si et seulement si : si chacun des membres d'une équation appartient à K (resp. à E) alors cette équation appartient à K (resp. à E).

Nous illustrons ce formalisme par un exemple [Gogu78].

Exemple 5.2 : Nous reprenons l'exemple 5.1 auquel on ajoute les cas d'erreur. Nous donnons un extrait de la signature et des équations de pile(ent) .

Signature de pile(ent) :

S = {p, e, b};
$\Sigma_{\lambda,p}$ = {VIDE},
$\Sigma_{\lambda,e}$ = Z,
$\Sigma_{\lambda,b}$ = {V, F},
$\Sigma_{p,p}$ = {DEPILER},
$\Sigma_{ep,p}$ = {EMPILER},
$\Sigma_{p,e}$ = {SOMMET},
$\Sigma_{p,b}$ = {ESTVIDE?},
$\Sigma_{ee,e}$ = {+, x, -};
$\Xi_{\lambda,e}$ = {SOMMET_ERR},
$\Xi_{\lambda,p}$ = {UNDERFLOW};

Equations

Ok-équations (K) :
DEPILER(EMPILER(E, P)) = P
SOMMET(EMPILER(E, P)) = E
ESTVIDE?(EMPILER(E, P)) = F
ESTVIDE?(VIDE)) = V
Plus les équations pour les opérations +, × et − sur Z

Err-équations (E) :
DEPILER(VIDE) = UNDERFLOW
SOMMET(VIDE) = SOMMET_ERR
DEPILER(UNDERFLOW) = UNDERFLOW

On obtient une présentation avec erreur P=(S, Σ, Ξ, K, E) pour pile(ent). Au regard de K et E, on peut remarquer que les erreurs sont propagées implicitement et qu'elles ne sont pas récupérées. ◇

Ce formalisme a l'avantage de séparer syntaxiquement les cas standards et les cas d'erreur. Par ailleurs, les exceptions sont propagées implicitement. Ceci est nécessaire si l'on ne veut pas écrire un trop grand nombre d'axiomes dans la spécification (si DEPILER(UNDERFLOW) est erronée, ne pas avoir à écrire que DEPILER(DEPILER(UNDERFLOW)) l'est aussi). Dans la partie suivante, nous étudions un formalisme prenant en compte la récupération des erreurs.

5.2.2 Les exceptions-algèbres

Une exception-algèbre [Bern86a][Bern86b] permet de distinguer les exceptions et les erreurs. La distinction repose sur le fait que la valeur d'un terme exceptionnel n'est pas nécessairement erronée (elle peut être Ok). Les propagations des erreurs et des exceptions sont implicites. L'algèbre d'exception permet de caractériser la propagation des erreurs, et la validation permet de caractériser la propagation implicite des exceptions. Des axiomes généralisés permettent de récupérer les exceptions.

Une exception-spécification est un 7-uplet SPEC = <S, Σ, L, Ok-Frm, Ok-Ax, Lbl-Ax, Gen-Ax>. <S, Σ, L> est une exception-signature Σ-exc où S est un ensemble fini de sortes, Σ est un ensemble fini d'opérations à domaine dans S, et L est un ensemble fini d'étiquettes d'exception. Ok_Frm est un ensemble fini de déclaration de formes Ok. Ok-Ax est un ensemble fini d'Ok-Axiomes. Un Ok-axiome est une équation conditionnelle entre termes Ok (une équation conditionnelle est de la forme *si prémisse alors équation*). Lbl-Ax est un ensemble fini d'axiomes d'étiquetage. Un axiome d'étiquetage est une équation éventuellement conditionnelle permettant d'associer une étiquette d'exception à certaines valeurs de l'algèbre. Enfin, Gen-Ax est un ensemble fini d'axiomes généralisés. Un axiome généralisé est une équation éventuellement conditionnelle qui permet de traiter les termes exceptionnels. Enfin, les exceptions ne se propagent pas si elles sont récupérées.

Les exceptions-algèbres décrivent la sémantique associée aux exceptions-spécifications. Une Σ-exc-algèbre, qui est une exception-algèbre sur une exception-signature, valide SPEC si et seulement si elle valide Ok-Frm, Ok-Ax, Lbl-Ax, et Gen-Ax.

- Une exception-algèbre A valide Ok-Frm si toutes les formes Ok ont une

valeur Ok dans A

- Elle valide Ok-Ax si deux termes équivalents sur Ok ont des valeurs égales après évaluation sur les valeurs de A.
- Elle valide Lbl-Ax, si pour tout axiome d'étiquetage :
  si les prémisses sont vérifiés dans A alors la conclusion l'est aussi
- Enfin, elle valide Gen-Ax, si pour tout axiome généralisé:
  si les prémisses sont vérifiés dans A, alors la conclusion l'est aussi

Une exception-algèbre est composée d'un sous-ensemble contenant les valeurs Ok et d'autant de sous-ensembles qu'il y a d'étiquettes d'exception. Un sous-ensemble associé à une étiquette d'exception contient toutes les valeurs étiquetées par elle. Nous exemplifions ce formalisme [Bern86a].

Exemple 5.3 : Il s'agit d'une spécification des piles bornées où les éléments placés dans la pile sont de sorte ELEM. Dans cet exemple, les piles bornées sont spécifiées comme une présentation au-dessus de NAT et d'une spécification prédéfinie ELEM contenant une sorte ELEM. La hauteur maximale des piles est *hauteur_max.*

S = {PILE}

$$
\begin{array}{rrcl}
\Sigma = & \text{VIDE} : & \rightarrow & \text{PILE} \\
& \text{EMPILER} : \ \text{ELEM PILE} & \rightarrow & \text{PILE} \\
& \text{DEPILER} : \ \text{PILE} & \rightarrow & \text{PILE} \\
& \text{SOMMET} : \ \text{PILE} & \rightarrow & \text{ELEM} \\
& \text{HAUTEUR} : \ \text{PILE} & \rightarrow & \text{NAT}
\end{array}
$$

L = { UNDERFLOW, OVERFLOW, PILEVIDE }

Ok-Frm =
$x_1 \in$ Ok-Frm & ... & $x_{hauteur\_max} \in$ Ok-Frm $\Rightarrow$
EMPILER(x_1, ..., EMPILER($x_{hauteur\_max}$, VIDE) $\in$ Ok-Frm
EMPILER(x, X) $\in$ Ok-Frm $\Rightarrow$ X $\in$ Ok-Frm

Ok-Ax :
DEPILER(EMPILER(x, X)) = X
SOMMET(EMPILER(x, X)) = x
HAUTEUR(VIDE) = 0
HAUTEUR(EMPILER(x, X)) = SUCC(HAUTEUR(X))

Lbl-Ax :
DEPILER(VIDE) $\in$ UNDERFLOW
SOMMET(VIDE) = PILEVIDE

X ∈ UNDERFLOW ⇒ DEPILER(X) ∈ UNDERFLOW
HAUTEUR(X) = hauteur_max ⇒ EMPILER(x, X) ∈ OVERFLOW
X ∈ OVERFLOW ⇒ EMPILER(x, X) ∈ OVERFLOW

Gen-Ax :
Il existe plusieurs traitements d'exceptions possibles pour les piles. Nous donnons des exemples d'axiomes pouvant figurer dans Gen-Ax. Si l'on veut nommer CRASH une pile UNDERFLOW, il suffit d'ajouter la constante CRASH dans la signature et d'écrire l'axiome généralisé suivant :

X ∈ UNDERFLOW ⇒ X = CRASH

Si l'on veut récupérer les piles UNDERFLOW sur la pile vide, il faut l'axiome :

X ∈ UNDERFLOW ⇒ X = VIDE

Si l'on veut récupérer l'application de EMPILER sur une pile pleine, en n'effectuant pas l'opération sur cette pile, l'axiome suivant est nécessaire :

HAUTEUR(X) = hauteur_max ⇒ EMPILER(x, X) = X

◇

Nous n'étudions pas plus en avant les formalismes permettant le traitement des exceptions dans les types de données. Il en existe de nombreux autres qui sont présentés dans [Bido83]. Dans le paragraphe suivant, nous étudions des langages de spécification algébrique prenant en compte le traitement d'exceptions.

5.3 Langages de spécification algébrique prenant en compte les exceptions

Nous présentons deux langages de spécification algébrique prenant en compte les exceptions. Nous voyons tout d'abord un langage de spécification fondé sur une distinction entre l'ensemble des valeurs correctes et l'ensemble des valeurs erronées. Ensuite, nous présentons un langage qui repose sur la caractérisation d'objets sûrs et non sûrs.

5.3.1 Un premier langage de spécification algébrique

Dans le langage de spécification algébrique PLUSS [Bido85], les valeurs correctes et erronées sont dans deux ensembles disjoints. Par conséquent, une opération qui peut produire une erreur a au moins deux co-domaines. Par exemple, dans la spécification d'une pile, l'opération POP admet deux co-domaines; STACK et STACK_Err. Il est possible de construire les modules de l'application à partir

de cette spécification. Tout d'abord, on doit en déduire un schéma de décomposition. Le schéma de décomposition d'un type donne la structure général d'un programme effectuant une opération sur une variable de ce type. Par exemple, un programme qui effectue une opération x sur un tableau peut traiter le premier élément puis le reste du tableau. Nous pouvons imaginer plusieurs autres structures pour un tel programme, aussi il existe plusieurs schémas de décomposition pour un type donné. La première étape de construction d'un programme consiste à choisir une ou plusieurs variables parmi les variables d'entrée du programme et à leur appliquer un des schémas de décomposition liés à leur type. Cette stratégie est appliquée récursivement sur les valeurs d'entrée restantes et les variables de sortie du schéma de décomposition. La deuxième étape consiste à écrire les corps des traitants. Enfin, on peut accroître la lisibilité du programme en appliquant des règles de transformation simples. Nous présentons un exemple qui montre les différentes étapes de la construction.

Exemple 5.4 : Il s'agit d'écrire la procédure COMBIEN qui compte le nombre d'éléments satisfaisant la propriété P dans un tableau. Nous donnons une spécification paramétrée de ARRAY :

`proc ARRAY(ELEM, INDEX) =`

`SORT` $Array, Array\_err$;

`FONCTIONS`

init	: $Index \times Index$	$\rightarrow Array \cup Array\_err$
illegal_modif	:	$\rightarrow Array\_err$
illegal_init	:	$\rightarrow Array\_err$
lwb	: $Array$	$\rightarrow Index$
upb	: $Array$	$\rightarrow Index$
_[_] := _	: $Array \times Index \times Elem$	$\rightarrow Array \cup Array\_err$;
_[_]	: $Array \times Index$	$\rightarrow Elem$
subarray	: $Array \times Index \times Index$	$\rightarrow Array \cup Array\_err$

`VARIABLES`

...

`AXIOMES`

...

$\{Array\text{-}err\}$
$i < lwb(t)$ *ou* $i > upb(t) \Rightarrow (t[i] := v) =$ illegal_modif;
$i > j \Rightarrow \text{init}(i, j) =$ illegal_init;

$\{Array\}$
$i \leq j \Rightarrow \text{init}(i, j)$

$i \geq \text{lwb}(t)$ *et* $i \leq \text{upb}(t) \Rightarrow t[i] := v$;
...

$\{Array \cup Array\text{-}err\}$
$n1 \leq n2$ *et* $j \leq n2 \Rightarrow$ subarray $(init(n1, n2), i, j) = \text{init}(i, j)$;
$i \geq n1$ *et* $i \leq n2 \Rightarrow \text{subarray}(t[i] := v, n1, n2) = \text{subarray}(t, n1, n2)[i] := v$;
$i < n1$ *ou* $i > n2 \Rightarrow \text{subarray}(t[i] := v, n1, n2) = \text{subarray}(t, n1, n2)$;

Nous expliquons l'axiome $i < lwb(t)$ *ou* $i > upb(t) \Rightarrow (t[i] := v) =$ illegal_modif. de $\{Array\_err\}$: i n'étant pas un index du tableau, la modification de $t[i]$ est impossible. Par conséquent, $t[i] := v$ est erronée. Nous présentons un schéma de décomposition du type Array (l'étape n est notée `(n)`):

```
SCHEMA SH1 {t:Array -> v:Elem, t1:Array}
   n1, n2, n3 : Index
   [n1 <- (1) : lwb(t);
    n2 <- (2) : upb(t);
    v <-  (3) : t[n1];
    n3 <- (4) : n1+1;
    t1 <- (5) : subarray(t, n3, n2)
   ]
  (5) -> illegal_init;
FIN SH1;
```

Ce schéma de décomposition définit la structure d'un programme qui effectue une opération sur un tableau. On obtient tout d'abord les bornes inférieures et supérieures de ce tableau. Ensuite, on extrait le premier élément et on obtient un deuxième tableau dont on a enlevé le premier élément. Nous détaillons les lignes du schémas. `lwb(t)` rend la borne inférieure du tableau et cette valeur est affectée à `n1`. `upb(t)` rend la borne supérieure du tableau. `n1` étant la borne inférieure du tableau, `v` est égal au premier élément du tableau et `n3` est l'indice de début du reste du tableau. `t1` est le reste du tableau. Enfin, les exceptions sont déduites de la spécification. Ici, l'exception *illegal_init* peut être signalée par l'opération `subarray`. La fonction COMBIEN après application du schéma de décomposition devient (nous notons `.../N/...` les parties de la fonction restant à compléter) :

```
FONCTION COMBIEN (t:IN Array) RETURN Nat IS SIGNALE erreur_ds_COMBIEN
   n1, n2, n3 : Index; v : Elem; t1 : Array;
   DEBUT
      n1:=lwb(t);
      n2:=upb(t);
      v:=t(n1);
      n3:=n1+1;
      t1:=subarray(t,n3,n2);
      .../1/...
```

```
   EXCEPTION
      QUAND illegal_init=>.../2/...
FIN COMBIEN;
```

Les traitants d'exceptions s'obtiennent directement à partir du schéma de décomposition. L'ensemble de variables obtenu comprend les variables de sortie du schéma, `t1` et `v`. Il faut donc introduire un appel récursif sur `COMBIEN(t1)` (en /1/) car `t1` est du même type que `t`. On obtient un nouveau traitant d'exception, qui est celui pour `erreur_ds_COMBIEN`. Le nouvel ensemble de variables est le résultat de l'appel récursif et la variable `v`. Pour terminer la première étape de construction, il suffit de tester si `v` satisfait le prédicat P. Enfin, il faut écrire les corps des traitants. On peut supprimer la détection de `erreur_ds_COMBIEN` car il faut la propager et la propagation est implicite. Le traitant de `illegal_init` (/2/) consiste à tester P sur `v`. Le code final de la fonction COMBIEN est :

```
FONCTION COMBIEN (t:IN Array) RETURN Nat IS SIGNALE erreur_ds_COMBIEN
   n1, n2, n3 : Index; v : Elem; t1 : Array; n4 : Nat;
   DEBUT
      n1:=lwb(t);
      n2:=upb(t);
      v:=t(n1);
      n3:=n1+1;
      t1:=subarray(t,n3,n2);
      n4:=COMBIEN(t1);
      SI P(v) ALORS RETURN n4+1 SINON RETURN n4 FSI;
   EXCEPTION
      QUAND illegal-init => SI P(v) ALORS RETURN(1) SINON 0 FSI;
FIN COMBIEN;
```

◇

Cette approche est intéressante car elle montre, à partir d'une spécification algébrique prenant en compte les exceptions, comment obtenir un programme tolérant les occurences d'exceptions. Nous pouvons remarquer que la propagation des exceptions est implicite. Ceci vient de la spécification algébrique. Nous avons vu qu'il serait trop laborieux de spécifier la propagation. Le deuxième langage que nous présentons repose sur l'utilisation d'une fonction de sécurité.

5.3.2 Un deuxième langage de spécification algébrique

Dans ce langage de spécification [Hore88], le traitement des exceptions est réalisé en utilisant des constructions syntaxiques qui distinguents les objets "sûrs" des objets "non sûrs". Un objet sûr est une valeur correcte et un objet non sûr est une valeur erronée. Des notations particulières ont été introduites dans la spécification. Elles permettent de désigner les termes des axiomes devant être sûrs ou non, et les constructeurs donnant des objets sûrs ou non. Pour récupérer une erreur,

il suffit d'écrire une opération. Des conditions de sécurité ont été introduites afin de décrire les types bornés. Elles indiquent dans quelle condition l'objet est sûr. Nous donnons un exemple.

Exemple 5.5 : Il s'agit de la spécification d'une pile bornée. Nous expliquons les notations au fur et à mesure de l'exemple, elles figurent entre les mots **co** et **fco**.

MODULE Pile_bornée;
...

CONSTRUCTEURS

co
Un constructeur suivi de "??" indique que le constructeur rend un objet non sûr. Dans notre exemple, une pile *Underflow* ou *Overflow* est erroné ou encore est un objet non sûr.
fco

Underflow → Pile_bornée ??;
Overflow → Pile_bornée ??;

co
Un constructeur suivi de "$$" indique que le constructeur rend un objet sûr.
fco

Nouvelle_Pile → Pile_bornée $$;

co
Un constructeur suivi d'une condition de sécurité entre "$" indique que le constructeur rend un objet sûr si la condition de sécurité est vérifiée.
Après l'opération *Empiler*, la pile est correcte, ou encore est un objet sûr, si sa hauteur est inférieure à 100.
Un terme précédé d'un "$" indique que l'axiome ne peut être appliqué que si le terme est un objet sûr.
L'opération *Empiler* ne peut avoit lieu que si la pile en argument est un objet sûr.
fco

Empiler → $ Pile_bornée × Nat → Pile_bornée $ Hauteur(Pile_bornée)< 100 $

OPERATIONS

Hauteur, Dépiler, Sommet, Est_nouvelle, Récupération;

co
Nous ne donnons pas les arités des opérations.
fco

DECLARATIONS

pb : Pile_bornée;
n : Nat;

Axiomes des constructeurs

co
Un terme précédé de "!" indique que l'axiome ne peut être appliqué que si le terme est un objet non sûr.
fco

Empiler(!pb,n)==pb;

co
Un terme précédé de "$" indique que l'axiome ne peut être appliqué que si le terme est un objet sûr .
Nous expliquons l'axiome qui suit :
Si après *Empiler*, le terme est un objet non sûr alors il vaut *Overflow*. En effet, cet objet est non sûr si la hauteur de la pile ($Empiler(pb, n)$) est supérieure ou égale à 100.Par ailleurs, *Empiler* ne peut avoir lieu qu'à la condition que la pile en argument soit un objet sûr.
fco

! Empiler($pb,n)==Overflow;

Axiomes des opérations

Hauteur(Nouvelle_Pile)==Zero;
Hauteur(Empiler(pb,n))==Succ(Longueur(pb));
Dépiler(Nouvelle_Pile))==Underflow;
...

co
Un axiome précédé de "$$" indique que l'axiome ne peut être appliqué que si tous ses arguments sont des objets sûrs.
L'axiome qui suit est équivalent à *Récupération($pb)==pb.*
fco

$$ Récupération(pb)==pb;

co
Un axiome précédé de "??" signifie que l'axiome ne peut être appliqué que si au moins un de ses arguments est un objet non sûr.
fco

?? Hauteur(pb)==NatErr;
?? Dépiler(pb)==pb;
...
?? Récupération(pb)==Nouvelle_Pile;

`FIN MODULE` Pile_bornée;

◇

Ce langage de spécification nous semble moins satisfaisant que le précédent car il introduit un grand nombre de notations, et les spécifications des cas erronés et exceptionnels sont mélangées. Il est aussi beaucoup moins simple. Dans le chapitre suivant, nous abordons brièvement le traitement d'exceptions dans le cadre de la programmation parallèle.

6 Traitement d'exceptions et parallélisme

Il existe peu de propositions de MTE dans le cadre de la programmation parallèle. Ces MTE sont souvent très proche de celui du langage ADA. Il s'agit généralement de signaler une exception vers un autre processus. Les exceptions peuvent être propagées entre les processus [Szal85], [Hals85]. Il existe une autre étude [Camp86] qui traite du signal d'une exception par plusieurs composants d'une action atomique. La solution retenue consiste à interrompre toutes les actions atomiques internes et à ne signaler qu'une exception qui regroupe toutes celles signalées. Dans ce chapitre, nous présentons un MTE pour des langages permettant l'exécution concurrente d'un nombre statiquement défini de commandes séquentielles [Bana89]. Nous introduisons tout d'abord le langage de base pour lequel le MTE est défini.

6.1 Le langage de base

Le MTE est défini au dessus du langage CSP [Hoar78]. Nous rappelons brièvement les principales caractéristiques de ce langage. La spécification de l'exécution concurrente de commandes séquentielles se fait au moyen d'une commande parallèle. Chaque composant de la commande parallèle est appelé *processus*. La communication entre les processus s'exprime au moyen de commandes d'émission et de réception. Une communication (*Rendez-vous*) survient quand un processus P nomme un processus Q comme destinataire de l'émission et que Q nomme

P comme source de la réception. Une commande d'émission (resp. réception) est notée *Destination!Expression* (resp. *Source?Valeur*). Les commandes gardées [Dijk75] permettent d'introduire le non determinisme. Les commandes d'émission et de réception (un CSP étendu est considéré) peuvent apparaitre dans les gardes. Par conséquent, la forme syntaxique la plus générale d'une garde est :

Expression booléenne; Commande d'émission-réception.

Si l'on suppose que toutes les expressions booléennes sont évaluées à faux, une commande gardée avec une commande d'émission ou de réception ne peut être selectionnée qu'à la condition que le processus nommé dans la commande d'émission (resp. de réception) soit prêt à exécuter la commande de réception (resp. d'émission) correspondante. Une commande gardée se note :

Garde → *Liste de commandes.*

La commande alternative s'écrit :

[*Garde* → *Liste de commandes* □ ... □ *Garde* → *Liste de commandes*].

Quand plusieurs gardes peuvent être selectionnées alors une seule est choisie, et ceci de manière indéterministe. La syntaxe de la commande répétitive est :

*[*Garde* → *Liste de commandes* □ ... □ *Garde* → *Liste de commandes*].

Cette commande termine quand pour toutes les gardes, l'expression booléenne est évaluée à faux et/ou le processus nommé dans la commande d'émission ou de réception est terminé. Nous définissons informellement le MTE dans le paragraphe suivant.

6.2 Présentation informelle du MTE

Le langage de programmation étendu avec le MTE proposé est appelé CSP$^+$. Un processus CSP$^+$ est un processus CSP auquel sont ajoutés des traitants. Les traitants sont définis au moyen d'une liste de traitants notée :

{ Exception$_1$: Traitant$_1$; ...; Exception$_n$: Traitant$_n$}

La liste de traitants définit implicitement les exceptions puisqu'elles sont nommées dans cette liste. Une liste de traitants définit les traitants pour la liste de commandes entre paranthèses immédiatement à sa gauche. Nous avons donc :

(Liste de commandes C){Traitants pour C}

Il existe deux types d'exceptions, les locales et les globales. Une exception locale n'est traitée que par le processus où elle est signalée. Une exception globale est traitée par tous les processus définissant un traitant global pour cette exception. La notation utilisée est la même pour les traitants locaux et globaux. La liste de traitants définit les traitants globaux quand elle est rattachée à la liste de commandes la plus externe du processus (corps du processus). Le fait qu'une exception soit locale ou globale est lié à son traitant; si le traitant est global alors l'exception l'est aussi. Les exceptions sont signalées au moyen de l'instruction RAISE qui permet aussi de passer des valeurs aux traitants.

Nous présentons le schéma de contrôle de ce MTE. Puisqu'il n'y a pas de concept de procédure ni de création dynamique de processus dans CSP, il n'y a pas d'association dynamique de traitants. Le MTE proposé est une extension du modèle terminaison. Le schéma de contrôle du traitement des exceptions locales est le même que celui de MODULA-2 (paragraphe 2.2.2). Les exceptions locales peuvent être propagées implicitement. Lorsqu'une exception globale e_g est signalée, le processus signalant doit être interrompu et les processus traitants avertis de l'occurence de e_g. Les processus traitants sont tous les processus dont la liste de traitants globale définit un traitant de e_g. Nous étudions la prise en compte d'une exception globale par un processus traitant. Si le corps du processus traitant est une commande répétitive, l'exception peut être traitée après chaque pas d'itération. Dans tous les autres cas, le signal sera pris en compte à la fin de l'exécution du corps du processus traitant ou lors d'une attente de rendez-vous avec le processus signalant. Cette dernière condition est nécessaire afin d'éviter un interblocage. Le processus signalant n'exécutera pas la commande d'émission ou de réception attendu par le processus traitant ce qui entraîne un blocage de ce processus. Nous décrivons le schéma de controle du processus signalant après exécution de l'instruction de signal. Si le corps du processus signalant est une commande répétitive alors l'étape courante de l'itération est abandonnée et l'exécution reprend à l'étape suivante. Si le corps du processus est une autre commande alors le processus est terminé. Le schéma de contrôle appliqué aux processus traitants, après exécution du traitant global, est le même. Si le processus traitant est une commande répétitive, alors la prochaine étape sera exécutée sinon le processus est terminé. Enfin, les processus étant exécutés en parallèle, plus d'une exception peut être signalée au même instant, le MTE assure que toute exception signalée sera traitée. Dans le paragraphe suivant, nous donnons quelques exemple de traces d'exécution.

6.3 Exemples de traces d'exécution

Nous illustrons le schéma de contrôle du MTE au moyen d'exemples simples. Nous appelons C_i, toute commande qui n'est pas une commande répétitive. La notation $(C_i)^-$ est utilisée pour exprimer l'exécution de C_i jusqu'à un point de signal; il s'agit d'une exécution partielle de C_i. Enfin, $(C_i)^*$ signifie que C_i est

exécuté n fois ($n \geq 0$).

Traitement d'exceptions local :

Soit le processus P suivant :

$P ::$
$[$
$(*[Cond_1 \rightarrow C_1 \square Cond_2 \rightarrow C_2])\ \{Exc_1 : Traitant_1;\ Exc_2 : Traitant_2\};$
$(C_3)\{Exc_3 : Traitant_3\};$
C_4
$]$

Tout d'abord, nous pouvons remarquer qu'il n'y a pas de traitants globaux déclarés dans P. Nous supposons que l'exception Exc_1 est signalée à l'étape i pendant l'exécution de C_1 ou C_2. Si C_{x_k} symbolise l'exécution de C_1 ou C_2 à l'étape k, la trace d'exécution de P est :

$$C_{x_1};\ ...;\ (C_{x_i})^{-};\ Traitant_1;\ C_{x_{i+1}};\ ...;\ C_{x_n};\ (C_3)^{-};\ Traitant_3;\ C_4$$

Traitement d'exceptions global :

Considérons un système de processus défini comme suit :

$Pr_1 :: (*[Cond_1 \rightarrow C_1])\{Exc_2 : Traitant_2\}||$
$Pr_2 :: (*[Cond_2 \rightarrow C_2])\ \{Exc1 : Traitant_1;\ Exc_2 : Traitant_2\}||$
$Pr_3 :: (C_3)\{Exc1 : Traitant_1\}$

Le point de signal de Exc_1 (resp. Exc_2) est quelque part dans C_1 (resp. C_2). Pr_1 ne communique pas avec Pr_3 mais, il communique avec Pr_2. Tous les traitants définis par les processus Pr_i sont globaux puisqu'ils sont rattachés aux listes de commandes les plus externes. Supposons que Pr_2 signale Exc_2, que Pr_1 signale Exc_1, et que Exc_2 soit traitée avant Exc_1, une trace d'exécution possible est :

Exécution de $Pr_1 : (C_1)^*;\ (C_1)^{-};\ Traitant_2;\ (C_1)^*;\ (C_1)^{-};\ (C_1)^*$
Exécution de $Pr_2 : (C_2)^*;\ (C_2)^{-};\ Traitant_2;\ (C_2)^*;\ (C_2)^{-};\ Traitant_1;\ (C_2)^*$
Exécution de $Pr_3 : (C_3);\ Traitant_1$

La première exécution partielle de C_1 est liée au rendez-vous entre Pr_1 et Pr_2; Exc_2 est détectée par Pr_1 au point de rendez-vous. La deuxième exécution partielle de C_1 est due au signal de Exc_1. Nous consacrons le paragraphe suivant à la sémantique axiomatique du MTE.

6.4 Un aperçu de la sémantique axiomatique

La notation utilisée est la même que celle introduite dans [Cris84] et présentée dans le paragraphe 4.2. Un bloc signalant peut être exécuté jusqu'á ce qu'il termine normalement ou jusqu'à ce qu'il signale une exception. Par conséquent, un bloc termine en un point x qui peut être égal soit à fin soit à une étiquette d'exception. Une règle axiomatique est de la forme :

$$P\,\{C\}\,x : Q$$

Cette règle signifie que si P est vrai avant l'exécution de C et si C termine au point x alors Q sera vrai après l'exécution de C. C est une commande figurant dans le processus que l'on nomme PR (pour *P*rocessus de *R*éférence) dans les règles. Pour être en mesure de prouver la correction des programmes CSP$^+$, le système proposé doit être complété par un système de preuve existant pour CSP, par exemple celui défini dans [Dijk75]. Nous donnons un aperçu du système d'axiomes en présentant deux de ses règles.

Traitement d'exceptions local :

Nous fournissons la règle définissant le traitement d'exception local quand le bloc signalant est une commande répétitive. Cette règle est appliquée quand le traitant ne signale aucune exception. Dans ce cas, les étapes suivantes de l'itération seront exécutées. Il s'agit d'une extension de la règle de l'itération donnée dans [Apt80]. Le prédicat Fin_j, $i \neq j, 1 \leq i,j \leq N$ est vrai si le processus PR suppose que le processus j est terminé. Ce prédicat est introduit pour définir la terminaison de l'itération. Le predicat $processus(S)$ est vrai si S est le corps du processus; il permet de ditinguer le traitement des exceptions globales du traitement des exceptions locales. Afin de simplifier l'écriture, nous écrivons $*[\Box_{k=1..g}\ G_k \rightarrow C_k]$ pour $*[G_1 \rightarrow C_1 \Box ... \Box G_g \rightarrow C_g]$.

$$\frac{\begin{array}{c} P \wedge b_k\{Pr_k\$x_k\}R_k, R_k\{C_k\}e_i : R'_i\ ,\ i \in [1,m]\ ,\ k \in [1,g], \\ R'_i\{H_i\}P, \\ \neg processus(*[\Box_{k=1,...,n} b_k; Pr_k\$x_k \rightarrow C_k]) \end{array}}{P\{(*[\Box_{k=1,...,n} b_k; Pr_k\$x_k \rightarrow C_k])\ \{e_1 : H_1; ...; e_m : H_m\}\}\ P \wedge \wedge_{j=1}^{g}(\neg b_j \vee Fin_j)}$$

Traitement d'exceptions global :

Cette règle s'applique aux processus dont le corps n'est pas une commande répétitive (cette commande est notée S). Le prédicat $Signal(Pr_i, Pr_j, e(u))$ est vrai si le processus Pr_j signale une exception globale e traitée par Pr_i.

1. $P\{S\}x : Q$,
2. $Q \wedge Signal(PR, Pr_j, e_i(u))\{e_i(v)\}\ R'_i \wedge \neg Signal(PR, Pr_j, e_i(u)), i \in [1,m], j \in [1,n]$,
3. $R'_i\{H_i\}y : Q, i \in [1,m]$,
4. $processus(S)$

$P\{(S)\{e_1 : H_1; ...; e_m : H_m\}\}Q \wedge \wedge_{i=1}^{m} \wedge_{j=1}^{n} \neg Signal(PR, Pr_j, e_i(u))$

Le prémisse 2 porte sur la valeur émise par le processus signalant (Pr_j) à PR. Les hypothèses locales faites sur cette valeurs sont controlées au moyen d'axiomes de communication lorsque l'on prouve la correction du programme. Les prémisses 2 et 3 concernent le traitement d'exceptions global. Le processus PR termine quand il n'y a plus d'exceptions globales à traiter ($\wedge_{i=1}^{m} \wedge_{j=1}^{n} \neg Signal(Pr_r, Pr_j, e_i(u))$ dans la postcondition). Nous concluons la présentation de ce MTE par un exemple

6.5 Exemple : Un système de serveurs de données coopérants

Cet exemple traite de la protection des données au moyen de la fragmentation et du codage. Une donnée, par exemple un fichier, est fragmentée en N parties qui sont distribuées sur N nœuds. De plus, une clé déduite de la donnée traitée est sur le site $N+1$. Chaque site peut être vu comme un serveur gardant une partie de la donnée. Quand un fragment est perdu, le serveur peut la recalculer en interrogeant les autres serveurs. En résumé, un fragment de donnée d_i peut être calculé au moyen de la fonction $Calculer\_fragment(d_1, ..., d_{i-1}, d_{i+1}, ..., d_N, \psi(d))$ où $\psi(d)$ est la valeur de la clé. La valeur de la donnée d est la concaténation de tous les d_i où $1 \leq i \leq N$. Nous donnons un exemple de fragmentation.

$N = 3$, et les octets sont successivement distribués sur les sites. La fonction ψ admet trois octets en paramètre et consiste en un ou exclusif. La fonction $Calculer\_fragment$ est aussi un ou exclusif. Si $d = 0111\ 1000\ 1101$ alors $d_1 = 0111$, $d_2 = 1000$, et $d_3 = 1101$. En appliquant un ou exclusif sur les d_i, on obtient $\psi(d) = 0010$. Par conséquent, chaque octet perdu peut être restauré en évaluant $Calculer\_fragment$ (xor) sur les fragments de la donnée disponible. Par exemple, $d_1 = d_2 \oplus d_3 \oplus \psi(d)$. Nous présentons la programmation de cet exemple.

Nous ne détaillons pas les requêtes traitées par les serveurs. Nous appelons les requêtes req_r et les traitements correspondants $Traitant\_req_r$. Les processus S_i sont les serveurs et le processus S_{N+1} est le "serveur de clé".

```
S(i : 1..N + 1) ::
        Erreur_signalée INIT False;
        ( *[□r : 1..nb_req NOT(Erreur_signalée); req_r → Traitant_req_r] )
        {
        Perte(site_id) :
                Donnée_dist(1..N) : Page;
                Reçue(1..N) : Boolean;
                k INIT 0;
                *[k < N → k := k + 1; reçue(k):=False ];
                [ site_id = i →
                        *[□∀ j ∈ {1..n+1}−{i} NOT(reçue(j)); S_j? Donnée_dist(j) →
                                reçue(j):=True;
                        ];
                        Donnée_locale := Calculer_fragment(Donnée_dist)
                □ site_id ≠ i →
                        [ Donnée_locale_présente → S_site_id! Donnée_locale
                        □ Donnée_locale_non_présente → RAISE Erreur]
                ];
        Erreur : Erreur_signalée := True;
        }
```

L'exception *Perte(Identité du site signalant)* peut être détectée durant l'exécution de $Traitant\_req_r$. Si le serveur ne peut accéder ses données, il signale l'exception avec son identité de site. L'exception *Erreur* est nécessaire puisqu'une donnée ne peut pas être restaurée si plus d'un fragment de donnée est inaccessible.

Nous n'étudions pas plus en avant ce MTE. Il nous semblait intéressant de le présenter car c'est une extension consistante d'un MTE séquentiel. Par ailleurs, il est simple et est défini formellement. Toutefois, ce MTE devrait être étendu de manière à prendre en compte la création dynamique de processus.

7 Conclusion

Nous avons étudié les structures de contrôle qui permettent de traiter les exceptions dans les programmes impératifs. Nous avons tout d'abord donné une définition informelle des MTE. Lorsqu'une opération détecte l'occurence d'une exception elle signale cette exception vers l'appelant. Une séquence d'instructions, appelée traitant, définie chez l'appelant est exécutée. Selon le modèle mis en œuvre par le MTE, l'opération signalante est soit abandonnée, soit reprise. Nous avons ensuite montré qu'un MTE est nécessaire dans un langage de programmation impératif pour traiter les exceptions. Le traitement des exceptions à l'aide de commandes conditionnelles n'est pas satisfaisant car les programmes sont moins structurés et moins lisibles que si l'on utilise un MTE. Les exceptions peuvent aussi être traitées à l'aide de procédures. Dans ce cas, la programmation est

plus complexe. La nécessité de définir une structure de contrôle particulière pour traiter les exceptions étant admise, nous avons proposé des critères d'évaluation d'un MTE. Nous retiendrons principalement les critères de simplicité et de correction.

La seconde partie de cette étude bibliographique est consacrée à la définition formelle des MTE. Nous avons premièrement proposé une définition dénotationnelle de deux MTE. La définition dénotationnelle d'un MTE utilise l'argument sémantique continuation qui permet de modéliser le contrôle. Il existe une continuation pour chaque cas exceptionnel. Cette continuation est composée du traitant suivi du reste du programme. Nous avons aussi étudié la définition axiomatique de MTE existants en nous intéressant plus particulièrement à l'aspect preuve de programme. Les différents cas exceptionnels figurent dans la spécification; la postcondition est la disjonction de l'effet du cas standard et du traitement des exceptions. La preuve de la correction du programme avec sa spécification consiste à vérifier que le programme signale les exceptions s'il y a occurence des conditions exceptionnelles et que les effets des traitants satisfont la postcondition.

Après avoir présenté le traitement des exceptions dans les programmes, nous nous sommes attardés sur le traitement des exceptions au niveau des données. Le traitement des exceptions dans les types abstraits algébriques permet notamment de spécifier les structures bornées telles que les tableaux ou les piles bornées. Dans un premier temps, nous avons exposé deux formalismes de traitement d'exceptions existants. Ils montrent tous les deux qu'il est possible d'avoir des spécifications lisibles avec traitement d'exceptions. Nous concluons cette partie par l'énoncé de deux langages de spécification algébrique.

Nous avons terminé cette étude par le traitement d'exceptions dans le cadre de la programmation parallèle. Si les propositions de MTE sont nombreuses et dans l'ensemble satisfaisantes dans le cadre de la programmation séquentielle, il n'en va pas de même lorsque l'on doit traiter les exceptions en présence de parallélisme. Nous pensons qu'il manque un MTE prenant en compte toutes les caractéristiques d'un langage de programmation parallèle.

Dans cette étude, nous avons principalement priviligié l'aspect définition formelle des MTE. C'est pourquoi nous avons délibérément écarté un sujet comme la mise en œuvre des MTE. Un MTE peut être mis en œuvre soit par réécriture des commandes du MTE en des commandes du langage de base, soit par modification du compilateur [Robe89]. La première approche a l'inconvénient de ne pas générer un code particulièrement efficace mais elle permet de ne pas remettre en cause la portabilité du compilateur. Par conséquent, le choix en faveur de l'une ou l'autre des approches repose sur ce que l'on attend du MTE. Il est même souhaitable de proposer les deux types de mise en œuvre.

8 Remerciements

Je tiens tout d'abord à remercier Jean-Pierre Banâtre pour m'avoir encouragée à écrire ce document. Je le remercie ainsi que Daniel Le Métayer pour leurs conseils et remarques qui m'ont considérablement aidés dans la rédaction de ce document.

Bibliographie

[Ada79] Preliminary Ada reference manual. *ACM Sigplan,* vol. 14, num. 6, June 1979.

[Apt80] K.R. Apt, N. Francez, W.P. De Roever. A proof system for communicating sequential processes. *ACM transactions on programming languages*, vol 2, num. 3, pp 359-385, July 1980.

[Bana81] J.P. Banâtre, B. Gamatie, F. Ployette. Traitement d'exceptions et de fautes résiduelles dans les langages de programmation. *RAIRO Informatique,* vol. 15, num. 1, pp. 3-38, 1981.

[Bana89] J.P. Banâtre, V. Issarny. An exception handling mechanism for communicating sequential processes : Design, Verification and Implementation. Soumis à publication.

[Bern86a] G. Bernot. *Une sémantique algébrique pour une spécification différenciée des exceptions et des erreurs; Application à l'implémentation et aux primitives de structuration des spécifications formelles,* Thèse de 3ème cycle, Paris-Sud Orsay, Février 1986.

[Bern86b] G. Bernot, M. Bidoit, C. Choppy. Abstract data types with exception handling: An initial approach based on distinction between exceptions and errors. *Theorical computer science,* vol. 46 , pp. 13-45, 1986.

[Best81] E. Best, F. Cristian. Systematic detection of exception occurences. *Science of computer programming,* vol. 1, num. 1, pp. 115-144, 1981.

[Bido83] M. Bidoit , M.C. Gaudel. *Spécification des cas d'exceptions dans les types abstraits algébriques : problèmes et perspectives,* num. 146, LRI, Orsay, 1983.

[Bido85] M. Bidoit, B. Biebow, M.C. Gaudel, C. Gresse, G.D. Guiho. Exception handling: Formal specification and systematic program construction . *IEEE transaction on software engineering,* vol. SE-11, num. 3, pp. 242-251, March 1985.

[Bret88] M. Bretz, J. Ebert. An exception handling construct for functional languages. *Proc. ESOP'88,* LNCS, pp. 160-174, Springer-Verlag, Nancy, March 1988.

[Camp86] R.H. Campbell, B. Randell. Error recovery in asynchronous systems. *IEEE transaction on software engineering,* vol. SE-12, num. 8, pp. 811-826, August 1986.

[Cocc82] N. Cocco, S. Dully. A mechanism for exception handling and its verification rules. *Computer language,* vol. 7, pp. 89-102, 1982.

[Cous88] G. Cousineau, G. Huet. *The CAML primer project,* Version 2.5, Projet formel INRIA-ENS, 1988

[Cris79] F. Cristian. *Traitement des exceptions dans les programmes modulaires* , Thèse de docteur ingénieur, Université Scientifique et Médicale de Grenoble, Septembre 1979.

[Cris82a] F. Cristian. Exception handling and software fault tolerance. *IEEE Trans. on computer science,* vol. C-31, num. 6, pp. 531-540, June 1982.

[Cris82b] F. Cristian. Robust data types. *Acta Informatica,* vol. 7, num. 9, pp. 365-397, October 1982.

[Cris83] F. Cristian. Reasonning about programs with exceptions. *Proceedings thirteenth international symposium on fault tolerant computing,* IEEE, ed., pp. 188-195, Milano, Italy, June 1983.

[Cris84] F. Cristian. Correct and robust programs. *IEEE transaction on software engineering,* vol. SE-10, num. 2, pp. 163-174, March 1984.

[Cris87] F. Cristian. Exception Handling *IBM Research Report 5724*, 1987.

[Dijk75] E.W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, vol 18, num 8, pp 453-457, August 1975.

[Gogu75] J.A. Goguen , J.W. Thatcher, E.G. Wagner, J.B. Wright. Abstract data types as initial algebras and correctness of data representation. *Conference on Computer Graphics, Pattern recognition and Data Structure,* pp. 89-93, 1975.

[Gogu78] J.A. Goguen. Abstract errors for abstract data types. *Formal description of programming concepts,* E.J. NEUHOLD, pp. 491, North-Holland, St. Andrews, N.B, Canada, 1978.

[Good75] J.B. Goodenough. Exception handling issues and a proposed notation. *Communications of the ACM* vol. 18, num. 12, pp. 683-696, December 1975.

[Hals85] R.H. Halstead Jr, J.R. Loaiza. Exception handling in multilisp. *IEEE,* pp. 822-830, 1985.

[Hoar78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM,* vol 18, num. 12, pp 666-674, August 1978.

[Hore88] I. Van Horebeek, J. Lewi, E. Bevers, L. Duponcheel, W. Van Puymbroeck. An exception handling method for constructive algebraic specification. *Software practice and experience,* vol. 18, num. 5, pp. 443-458, May 1988.

[Knud87] J.L. Knudsen. Better exception handling in block structured systems. *IEEE Software,* vol. 17, num. 2, pp. 40-49, May 1987.

[Levi77] R. Levin. *Program structures for exceptional condition handling,* Thèse, Carnegie-Mellon University, June 1977.

[Lisk79] B. Liskov, A. Snyder. Exception handling in CLU. *IEEE transaction on software engineering,* vol. SE-5, num. 6, pp. 546-558, November 1979.

[Lukh80] D.C. Lukham, W. Polak. Ada exception handling : An axiomatic approach. *ACM transactions on programming languages and systems,* vol. 2, num. 2, pp. 225-233, April 1980.

[Mell77] P.M. Melliar-Smith, B. Randell. Software Reliability : The role of programmed exception handling. *SIGPLAN notices,* vol. 12, num. 3, pp. 95-100, March 1977.

[Miln84] R. Milner. A proposal for standard ML. *ACM conference record of the 1984 symposium on LISP and functional programming,* pp. 184-197, 1984.

[Robe89] E.S. Roberts. *Implementing exceptions in C,* Digital Systems Research Center, Palo Alto, Ca , 1989.

[Rovn85] P. Rovner, R. Levin, J. Wick. *On extending MODULA-2 for building large, integrated systems,* Digital Systems Research Center, Palo Alto, Ca , January 1985.

[Schm86] D.A. Schmidt. *Denotational semantics : A methodology for language development,* Allyn and Bacon, INC., 1986.

[Schw79] R.L. Schwartz. An axiomatic treatment of ALGOL 68 routines. *Proceedings Sixth International Conference on Automata, Languages and Programming (Graz, Australia)* , pp. 530-545, Springer Verlag, Berlin, July 1979.

[Szal85] A. Szalas, D. Szczepanska. Exception handling in parallel computations. *SIGPLAN notices,* vol. 20, num. 10, pp. 95-104, October 1985.

[Yemi85] S. Yemini, D.M. Berry. A modular verifiable exception handling mechanism. *ACM transactions on programming languages and systems,* vol. 7, num. 2, pp. 214-243, April 1985.

[Yemi87] S. Yemini, D.M. Berry. An axiomatic treatment of exception handling in an expression oriented language. *ACM transactions on programming languages and systems,* vol. 9, num. 3, pp. 390-407, July 1987.

LISTE DES DERNIERES PUBLICATIONS INTERNES

PI 490 **THE SIGNAL SOFTWARE ENVIRONMENT FOR REAL-TIME SYSTEM SPECIFICATION, DESIGN, AND IMPLEMENTATION**
Albert BENVENISTE, Paul LE GUERNIC, Christian JACQUEMOT
34 Pages, Septembre 1989.

PI 491 **PHYSIQUE QUALITATIVE : PRESENTATION ET COMMENTAIRES**
Qinghua ZHANG
48 Pages, Septembre 1989.

PI 492 **SPARSE MATRIX MULTIPLICATION ON VECTOR COMPUTERS**
Jocelyne ERHEL
20 Pages, Septembre 1989.

PI 493 **THE SUPERIMPOSITION OF ESTELLE PROGRAMS : A TOOL FOR THE IMPLEMENTATION OF OBSERVATION AND CONTROL ALGORITHMS**
Benoît CAILLAUD
30 Pages, Septembre 1989.

PI 494 **EMPLOI DU TEMPS : PROBLEME MATHEMATIQUE OU PROBLEME POUR LA PROGRAMMATION EN LOGIQUE AVEC CONTRAINTES**
Xavier COUSIN
64 Pages, Septembre 1989.

PI 495 **NUMERICAL METHODS IN MARKOV CHAIN MODELING**
Bernard PHILIPPE, Youcef SAAD, William J. STEWART
46 Pages, Septembre 1989.

PI 496 **PLANIFICATION EN UNIVERS MONO ET MULTI-AGENTS (Définitions, concepts, objectifs, présentation d'un planificateur)**
Philippe PORTEJOIE
92 Pages, Octobre 1989.

PI 497 **LE TRAITEMENT D'EXCEPTIONS - ASPECTS THEORIQUES ET PRATIQUES**
Valérie ISSARNY
80 Pages, Octobre 1989.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique